GENERIC ELEMENT PROCESSOR
(APPLICATION TO NONLINEAR ANALYSIS)

by

Gary Stanley
Lockheed Palo Alto Research Laboratory
Computational Mechanics Section

ABSTRACT

The CSM software Testbed, currently under joint development by NASA Langley (LaRC), Lockheed Palo Alto Research Laboratory (LPARL) and selected University Grantees, will provide researchers with a common workbench to develop computational methods for advanced structural analysis problems (such as the NASP and Space Station); and to apply these methods to large-scale components without major software implementation changes. The most novel aspect of this Testbed is that it is designed to sustain both parallel development (by Government, University and Industry) and parallel-processing computers.

The focus ~~of the present talk~~ is on one aspect of the ~~CSM~~ Testbed: finite element technology. The approach involves a "Generic Element Processor": a command-driven, database-oriented software shell that facilitates introduction of new elements into the testbed. This "shell" features an element-independent corotational capability that upgrades linear elements to geometrically nonlinear analysis, and corrects the rigid-body errors that plague many contemporary plate and shell elements. Specific elements that have been implemented in the Testbed via this mechanism include the Assumed Natural-coordinate Strain (ANS) shell elements, developed with Professor K.C. Park (University of Colorado, Boulder), a new class of curved hybrid shell elements, developed by Dr. David Kang of LPARL (formerly a student of Professor T. Pian), other shell and solid hybrid elements developed by NASA personnel, and recently a repackaged version of the workhorse shell element used in the traditional STAGS nonlinear shell analysis code.

The presentation covers (i) user and developer interfaces to the generic element processor, (ii) an explanation of the built-in corotational option, (iii) a description of some of the shell-elements currently implemented, and (iv) application to sample nonlinear shell postbuckling problems. There will also be a brief overview of the CSM Testbed: its current status, future directions, and potential benefits for all involved with the CSM effort.

OUTLINE

- PROBLEM: CSM ("Chaotic" Structural Mechanics)
- APPROACH: The Role of the Testbed
- PROGRESS: Testbed Development Overview
- GENERIC ELEMENT PROCESSOR
  - — Why? (Motivation)
  - — What? (User's View; Developer's View)
  - — How? (Nonlinear Applications)
- CONCLUSIONS
  - — Summary
  - — Pitfalls
  - — Plans

OUTLINE

The presentation begins with a curious look at the state of this so-called "mature" technology called computational structural mechanics (CSM). After motivating the need for an end to the present chaos, the CSM Testbed is presented as a viable solution that will enable us to approach next generation structural problems in concert. Next, an overview of the Testbed's development is presented, as background for the recent development of a Generic Element Processor. The Generic Element Processor is then discussed from all angles: why it was developed, how it should make life easier for the user, the algorithm developer, the element developer; and, finally, a demonstration of its application to nonlinear shell postbuckling analysis. The talk concludes with a summary of progress, a list of pitfalls, and specific plans for the next stage of development.

"Chaotic" Structural Mechanics (1)

FINITE ELEMENTS PROVIDE CONVENIENT BUILDING BLOCKS

but:

- Why are there hundreds of different types?
- Why does each have a pathology?
- How do we select an element for an application?
- How do we adaptively refine a model — for nonlinear analysis?

"Chaotic" Structural Mechanics (1)

Finite elements have been successful for very good reasons. One is that they provide convenient building blocks — both conceptually and in software — for general purpose structural analysis. However, why do we have hundreds of different element types, when only a few intrinsic types are needed (e.g., beam, shell, solid, constraint)? Also with the plethora of different variations on a theme, it is disturbing that all elements seem to have at least one pathology (e.g., locking, spurious modes, distortion sensitivity). Thus, the *number* of different elements is not necessarily a sign of maturity. Given this situation, it becomes quite difficult to select an element for a given application. Finally, while techniques for discrete error estimation and adaptive mesh refinement are becoming a reality, there is a long way to go before such techniques can be applied to nonlinear analysis.

"Chaotic" Structural Mechanics (2)

NONLINEAR SOLN ALGORITHMS CAN TRAVERSE LIMIT POINTS

but:

- Why do they get stuck at bifurcations?
- Why are some sensitive to physical units?
- How do we pick an error tolerance?
- How do we parallelize them?
- How many iterations will it take to implement?

"Chaotic" Structural Mechanics (2)

Another CSM area where significant progress has been made is nonlinear solution algorithms. Static continuation methods, such as arc length control, now make it possible to traverse limit points with ease. However, most practical shell problems involve bifurcations, which can lead to a variety of potential equilibrium solutions, and simple continuation methods do not provide a means for finding such paths. While research in this area is progressing rapidly, results are embryonic and there is some indication that *dynamic* analysis (or dynamic relaxation) may be the only reliable way to solve complex shell postbuckling problems. More work is also needed to make the static continuation methods now in use more robust. For example, the success of some algorithms may actually depend on the choice of physical units — due to scaling problems between displacements and forces and/or between translational and rotational degrees of freedom. Moreover, the selection of an error tolerance for nonlinear convergence remains a "black art"; in fact even the definition of an error norm can be plagued with the same kind of scaling problems as arise for continuation methods. Finally, we are just beginning to look at algorithms for solving nonlinear problems faster — on parallel computers. This may may add yet another dimension to the reliability hurdles mentioned above.

"Chaotic" Structural Mechanics (3)

"GENERAL PURPOSE" F.E. PROGRAMS ABOUND!

but:

- Why is there always something missing?
- Why is it so hard to add it?
- Why must we pay to use black boxes?
- Why can't we use the same code for research AND production?
- How will they survive the parallel processing era?

"Chaotic" Structural Mechanics (3)

The last indicator of the "state of our art" to be looked at here is the availability of CSM software. While general-purpose structural finite element programs abound (and to a lesser extent those based on boundary elements, finite differences, etc.) there is little indication that they are doing anything to advance technology. They represent an assortment of black boxes, with something missing from each, and usually require great difficulty to add anything new. What's more, we are forced to pay for these technological straight jackets — or to develop our own private software at much greater cost. Haven't we learned enough during the past two decades to develop modular, extendible, white-box software that can be used by both researchers *and* production analysts?

The Role of the CSM Testbed

Lockheed Palo Alto *CSM Testbed Development*

An

EXTENDIBLE

PUBLIC DOMAIN

TOOL KIT

for

PARALLEL DEVELOPMENT

and

PARALLEL PROCESSING

of

ADVANCED CSM SOLUTIONS

The Role of the CSM Testbed

The CSM Testbed is an approach for obtaining advanced solution to advanced structural mechanics problems by overcoming the obstacles described in the preceeding charts. It is an extendible, public domain software tool kit (or "operating system" if you like) designed for both parallel development and parallel processing, and for applications ranging from academic research on new analysis methods to production analysis of complex aerospace vehicles. The next few charts describe its evolution over the past few years from a set of independent software utilities to what is approaching an integrated programming environment.

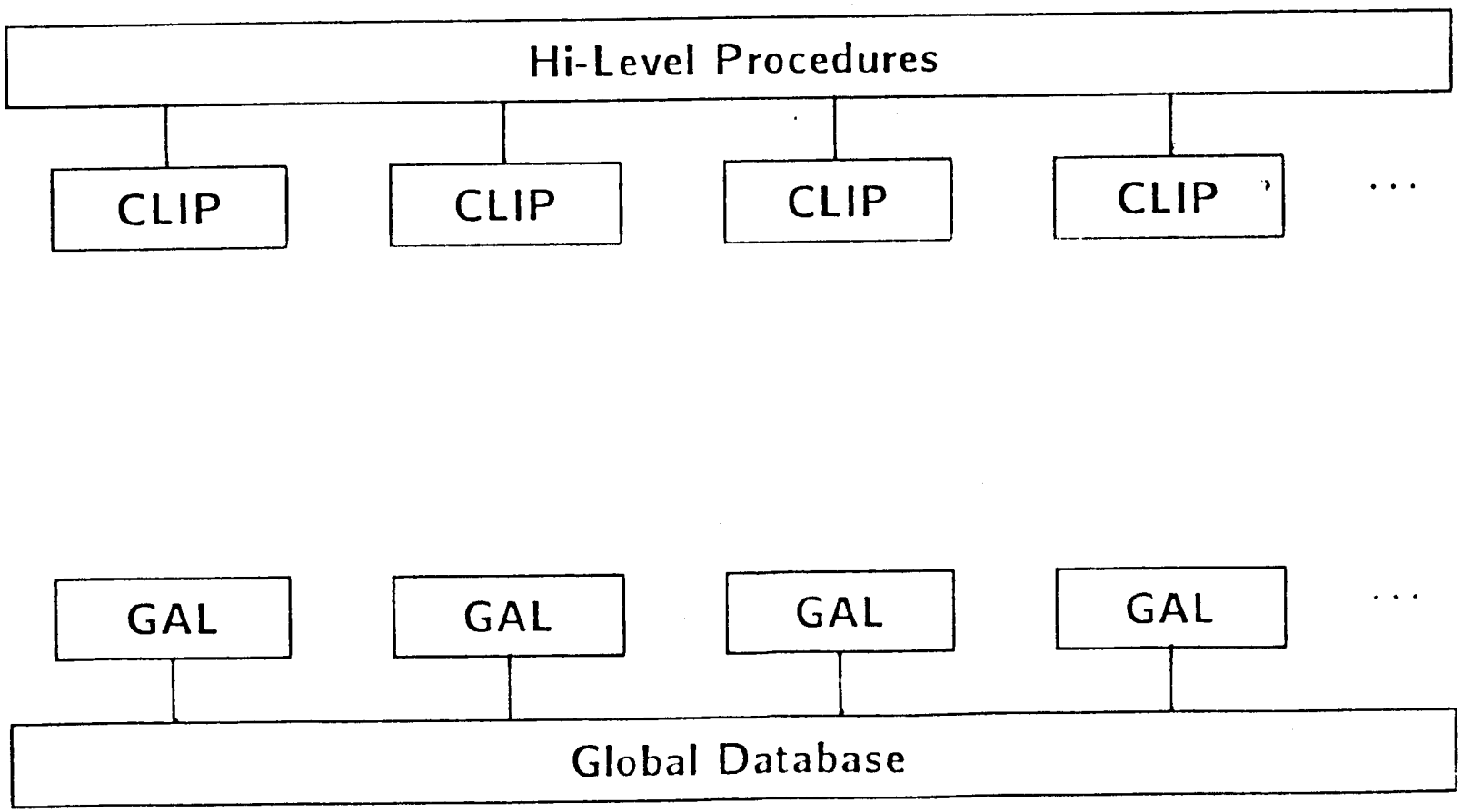
TESTBED DEVELOPMENT OVERVIEW

- PHASE 0 — NICE and SPAR
- PHASE 1 — NICE/SPAR
- PHASE 2 — CSM

TESTBED DEVELOPMENT OVERVIEW

The CSM Testbed began as a separate collection of software architecture utilities (NICE) and finite element analysis processors (SPAR). During the past several years, NICE and SPAR have been merged into the prototype system: NICE/SPAR. Since then, additional capabilities have been added, as well as additional layers of architecture to *facilitate* the addition of new capabilities. It is an ongoing process, which will culminate over the next few years in a fully integrated, yet distributed, Testbed system that we will refer to here simply as CSM.

PHASE 0: NICE
(Architecture Tools)

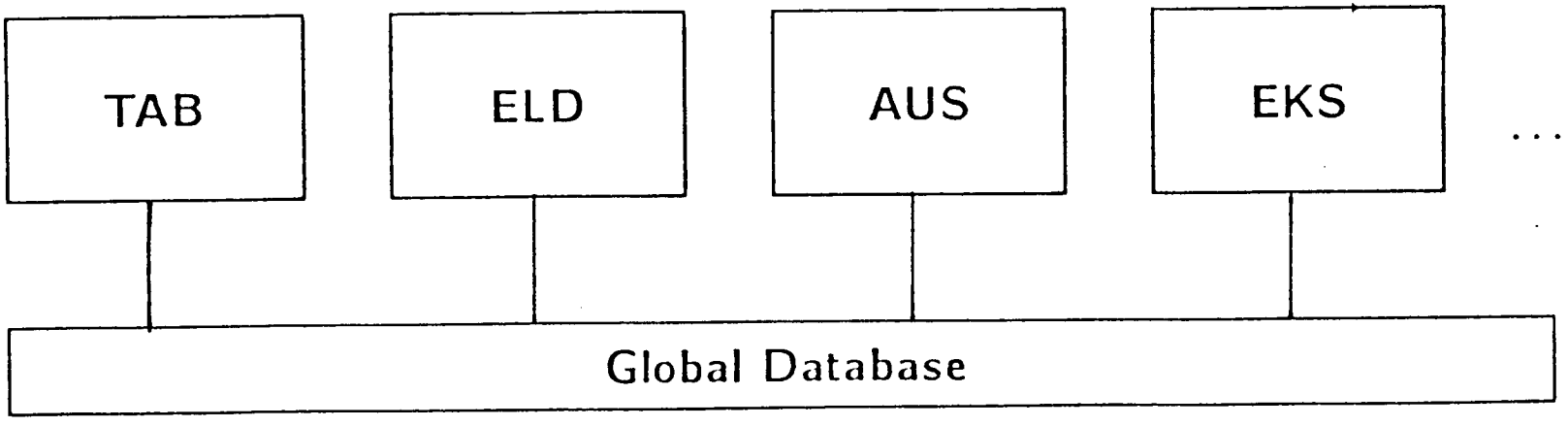


PHASE 0: NICE
(Architecture Tools)

Lockheed Palo Alto *CSM Testbed Development*

NICE (Network of Integrated Computational Elements) can be viewed as a set of software architectural utilities for CSM [1]. These utilities may be partitioned into those for database management (GAL) and those for command language/procedure interpretation (CLIP). Both CLIP and GAL may be replicated, via an object library, within an arbitrary number of independent software Processors (i.e., executables), thereby forming a computational network. The beauty of a network constructed with the NICE architecture is that all Processors are forced to communicate formally through a global database, and all Processors may be conveniently controlled through a high-level command/procedure language (called CLAMP — Command Language for Applied Mechanics Processors).

PHASE 0: SPAR
(Linear Analysis)

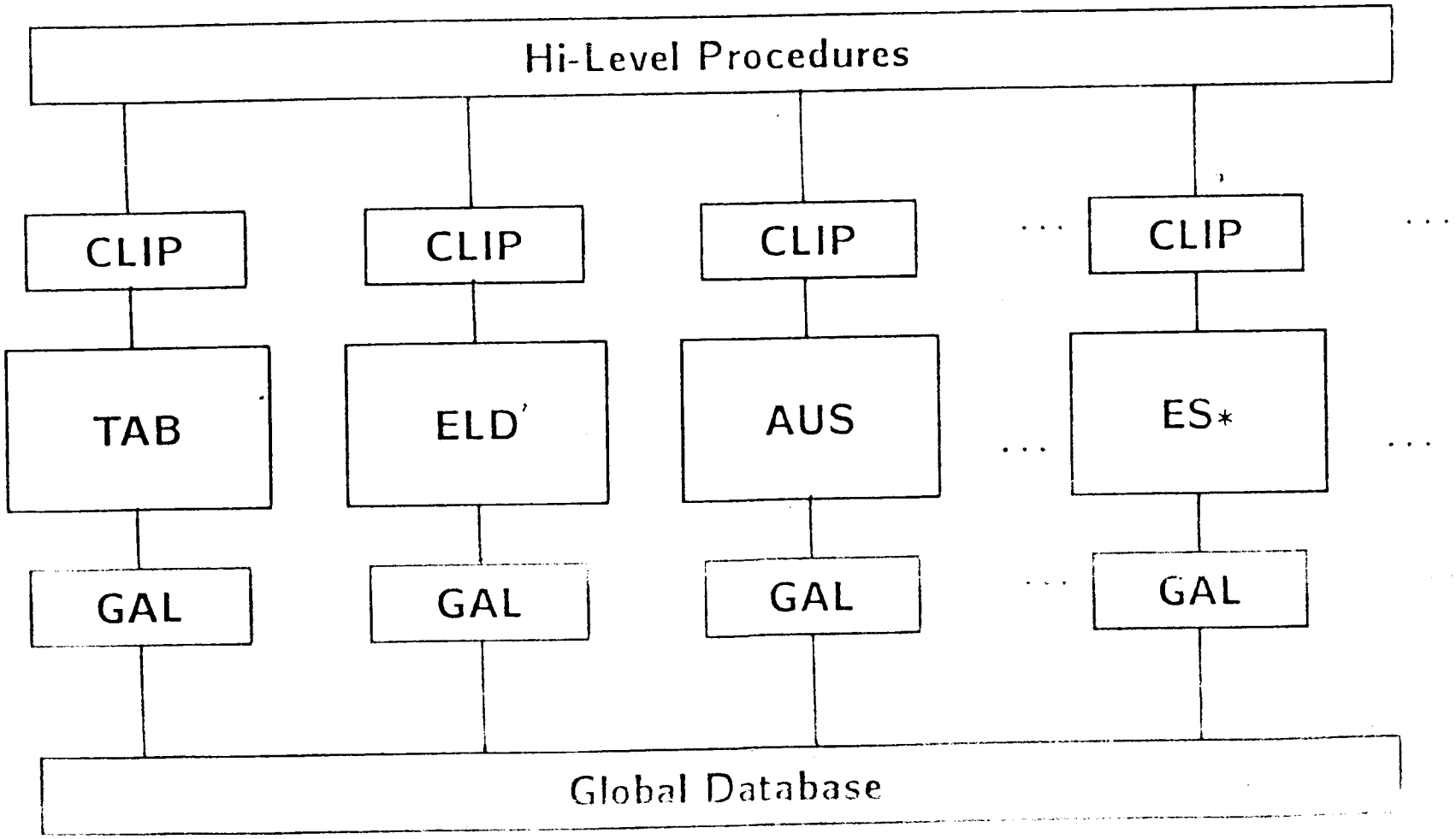


PHASE 0: SPAR
(Linear Analysis)

Lockheed Palo Alto *CSM Testbed Development*

SPAR may be viewed as a set of independent software Processors which collectively perform linear finite element analysis [2]. The beauty of SPAR is its modularity, and the fact that it employs a global database for interprocessor communication. However, while it employs a rudimentary command language for each Processor, SPAR (by itself) does not provide a Procedure capability. Hence, runstreams are restricted to sequential operation of Processors. This and the lack of appropriate Processors to perform nonlinear functions has limited the usefulness of SPAR in the past. Yet it has gained a certain measure of reliability and familiarity over the years (especially at NASA) and has remained in the *public domain*, which made it an excellent starting point.

PHASE 1: NICE/SPAR
(Prototype)

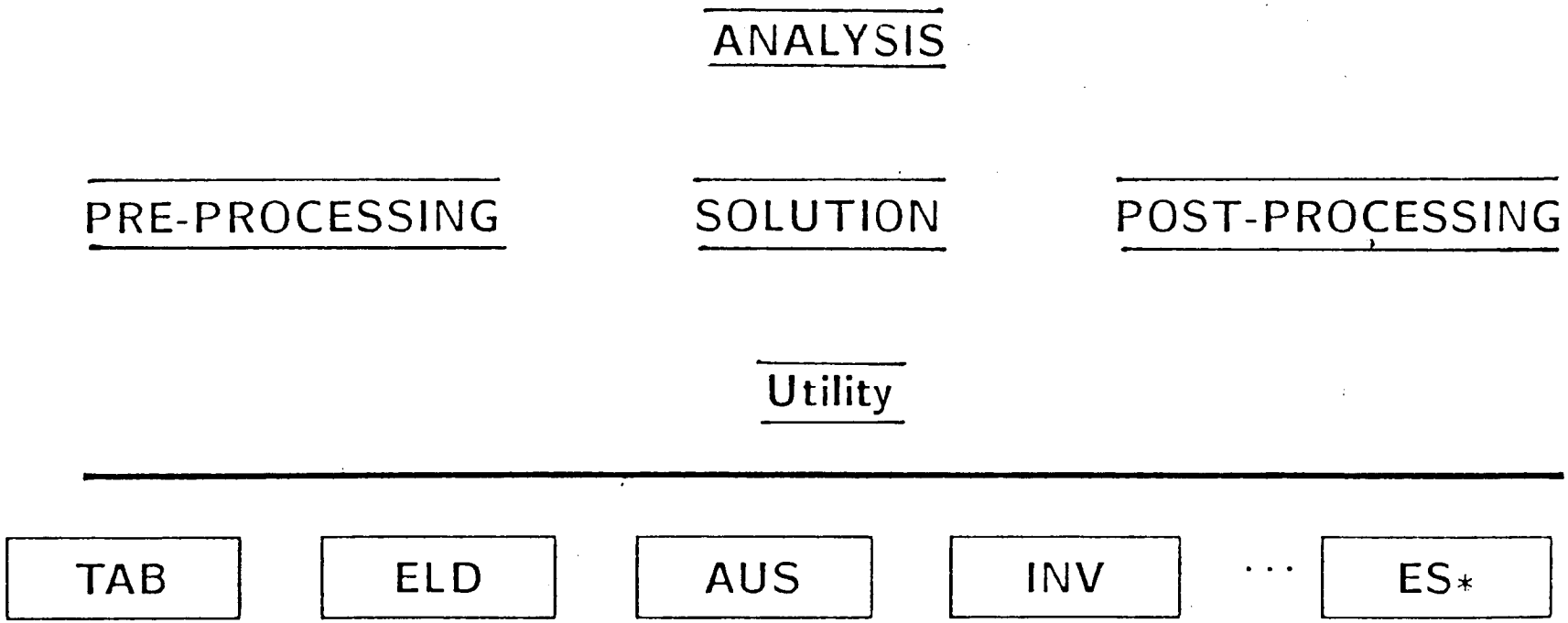


PHASE 1: NICE/SPAR
(Prototype)

An obvious match, NICE and SPAR were mated by interfacing SPAR's database management utilities to NICE/GAL [3], and its command language interpretation utilities to NICE/CLIP [4]. In so doing, SPAR gained a high-level command Procedure capabilitiy and a more flexible (multi level) database. Note also that as a part of the merger, LPARL developed some additional analysis Processors and Procedures that upgraded the system from a linear code to one with advanced geometrically nonlinear analysis capabilities (e.g., to handle shell postbuckling and multi-body dynamics problems). The generic element Processor, ES*, and various specific element implementations constructed with it, are a sample of LPARL's contributions.

NICE/SPAR Hi-Level Procedures

Lockheed Palo Alto *CSM Testbed Development*



NICE/SPAR Hi-Level Procedures

The next few charts illustrate how the NICE Procedure level was exploited in NICE/SPAR to conceal the details of individual Processor execution, permiting the user to access the system with an appropriate level of abstraction. Thus, very high-level Analysis Procedures were written (in the CLAMP language) which in-turn invoke generic model (Pre-processing) Procedures, linear and nonlinear Solution Procedures, and an assortment of Post-processing Procedures (as shown in this chart). Note that direct execution of individual Processors (shown below the horizontal line) is still permitted; high-level Procedures simply provide a tool that can be quickly and easily tailored to the application — be it research- or production-driven.

NICE/SPAR

Pre-Processing Procedures


```
*procedure GEN_CYLINDER (
                elt_proc = ES1 ;  elt_type = ANS9 ;  elt_pars = 0.0
                radius = 1.0 ; thickness = .1 ; E = 1.e7 ; PR = .3
                bc_procedure = PINCHED_CYL_BC
                nel_x = 3 ; nel_y = 3 )

    [xqt TAB
    *call ES ( function = 'DEFINE ELEMENTS' )
    [xqt ELD
    [xqt AUS
    [xqt E
    [xqt TOPO
*end
```

NICE/SPAR

Pre-Processing Procedures

Lockheed Palo Alto *CSM Testbed Development*

Shown in this chart is an example (in outline form) of a CLAMP Procedure to define a generic cylindrical shell finite element model, using NICE/SPAR Processors. The example merely illustrates the nature of Procedure arguments (they are name-lists, allow default settings and are order-independent), shows the incorporation of SPAR Processor executions (e.g., [xqt TAB], and also shows the invocation of one Procedure from another: in this case, GEN_CYLINDER calls ES, which, as we shall see, is a multi-purpose window to the Generic Element Processor.

NICE/SPAR

Linear Solution Procedures


```
*procedure L_STATIC_1 ( )

  *call ES ( function = 'FORM STIFFNESS/MATL' )
  [xqt K
  [xqt INV
  [xqt SSOL
 :
*end
```

NICE/SPAR

Linear Solution Procedures

Lockheed Palo Alto *CSM Testbed Development*

In this chart, an example of the simplest form of solution Procedure is shown: linear statics. Here, the ES Procedure is invoked to form all element stiffness matrices; Processor K is used to assemble the global stiffness matrix; INV is used to factor it; and SSOL is used to obtain a solution via forward/backward substitution. Subsequently, stresses may be computed, reactions checked, iterative refinement performed, etc.

NICE/SPAR

Nonlinear Solution Procedures

Lockheed Palo Alto *CSM Testbed Development*

```
*procedure NL_STATIC_1 ( beg_step = 1 ; beg_load = .1
                         max_step = 10 ; max_load = 1.0
                         max_iters = 7 ; max_cuts = 3
                         error_tol = 1.e-4 ; arc_scale = 1.0
                         nl_geom = 2 ; corotation = <true> )

  *do step = [beg_step], [max_step]

    *call STIFFNESS ( disp = <d_np1_i> )

    *do iter = 1, [max_iters]

      *call FORCE ( disp = <d_np1_i>; force = <r_np1_i> )

      *call SOLVE ( rhs = <r_np1_i> ; soln = <d_inc> )

      [xqt VEC
          <d_np1_ip1>  <-  <d_np1_i>  +  <d_inc>
          ROTATE <T_np1_i> * <d_inc>  ->  <T_np1_ip1>
```

NICE/SPAR

Nonlinear Solution Procedures

Lockheed Palo Alto *CSM Testbed Development*

This chart shows a very brief *excerpt* of a nonlinear static solution Procedure written for NICE/SPAR. The actual Procedure employs Crisfield's version of Rik's arc-length control algorithm, which is an adaptive load-stepping strategy for traversing complex curves in load/displacement space. At the core of the Procedure is the traditional iterative/incremental modified Newton-Raphson algorithm, with an arc-length constraint equation appended to determine the load increment as an additional unknown. Note that the Procedure argument list includes a number of strategy parameters (only a subset are shown) and involves DO loops, calls to other Procedures and direct Processor executions. In particular, notice the boxed section which illustrates one of many potential matrix-algebra oriented command languages. In this case the matrix language is indigenous to the VEC Processor, and is used to update the system displacement vector and nodal rotation triads at a given nonlinear iteration.

It is one of our goals to jointly develop a *standard/extendible* matrix language, and to make its interpretation an intrinsic part of the *architecture*. This would make procedure-writing more consistent, more natural and hence more straightforward.

Finally, note that no reference is made in Procedure NL_STATIC_1 to the Generic Element *Procedure*: ES. Instead, the ES calls are covered with an even higher level of Procedures, such as STIFFNESS and FORCE, which form the element arrays (via ES) and assemble them into system arrays as well.

NICE/SPAR PROCEDURES

ANALYSIS

Lockheed Palo Alto *CSM Testbed Development*

```
*procedure HINGED_CYLINDER ( elt_proc = ES1; elt_type = ANS9 ;
                             nel_x = 7 ; nel_y = 7 )

  *call GEN_CYLINDER ( radius = 100.; nel_x = [nel_x]; nel_y = [nel_y] ...)

  *call NL_STATIC_1 ( MAX_STEP = 20 )

*end
```

```
$ CSM
  *call HINGED_CYLINDER ( nel_y = 10; nel_x = 10 )
```

NICE/SPAR PROCEDURES

ANALYSIS

Lockheed Palo Alto *CSM Testbed Development*

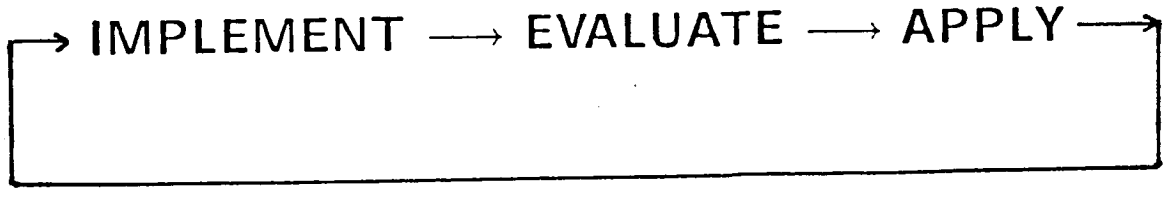
The analyst with a problem to solve over and over again doesn't want to be bothered with the details of integrating pre-processing, solution and post-processing Procedures together each time. Hence, it is convenient to develop top-level Analysis Procedures for specific problems; with only those parameters that are volatile — i.e., those that the analyst is interested in changing — exposed. A trivial example of such an Analysis Procedure is shown in this view: a Procedure to analyze the classical hinged cylinder benchmark problem. Note that such Procedures are easy to write, and facilitate automatic parameter studies and/or standard quality assurance testing. A typical invocation of the Analysis Procedure is shown at the bottom of the chart, in which the command CSM is used to start-up the Testbed. Note that only the mesh parameters, `nel_x` and `nel_y`, are changed (from their default settings) by the call to Procedure `HINGED_CYLINDER`.

GENERIC ELEMENT PROCESSOR

Motivation

- Adding new elements to the Testbed is difficult !!

- "Experimental Element" facility is inadequate — LINEAR only
- Testbed should provide common GAUNTLET (strainer)
- New elements should be easy to:



GENERIC ELEMENT PROCESSOR
MOTIVATION

The preceding charts illustrate how useful Procedures can be for adding *some* new capabilities to the Testbed. However, adding new elements to the Testbed has been quite difficult. One reason is due to the way the original SPAR element Processors are organized. Each element Processor performs a single function for *all* element types (e.g., EKS forms all element linear stiffness matrices). And there are many element functions (e.g., geometric stiffness – KG, consistent mass – M, stress and internal force - GSF, etc.). Hence, to add a new element one had to become familiar with each of these executive programs. Also, these SPAR Processors distinguish between "standard" elements, which are built-in and enjoy complete functionality; and "experimental" elements, which are considered temporary and which have only some of the complete set of element functions. Moreover, while adding an "experimental element" is relatively easy, converting it to a "standard element" is extremely difficult (this is compounded by the lack of in-line documentation within the SPAR element Processors). Finally, since the SPAR element Processors were designed for linear analysis only, it appeared that an ad hoc attempt to extend their capabilities to nonlinear analysis would have resulted in tangled, unmaintainable coding — i.e., it would have been a kluge.

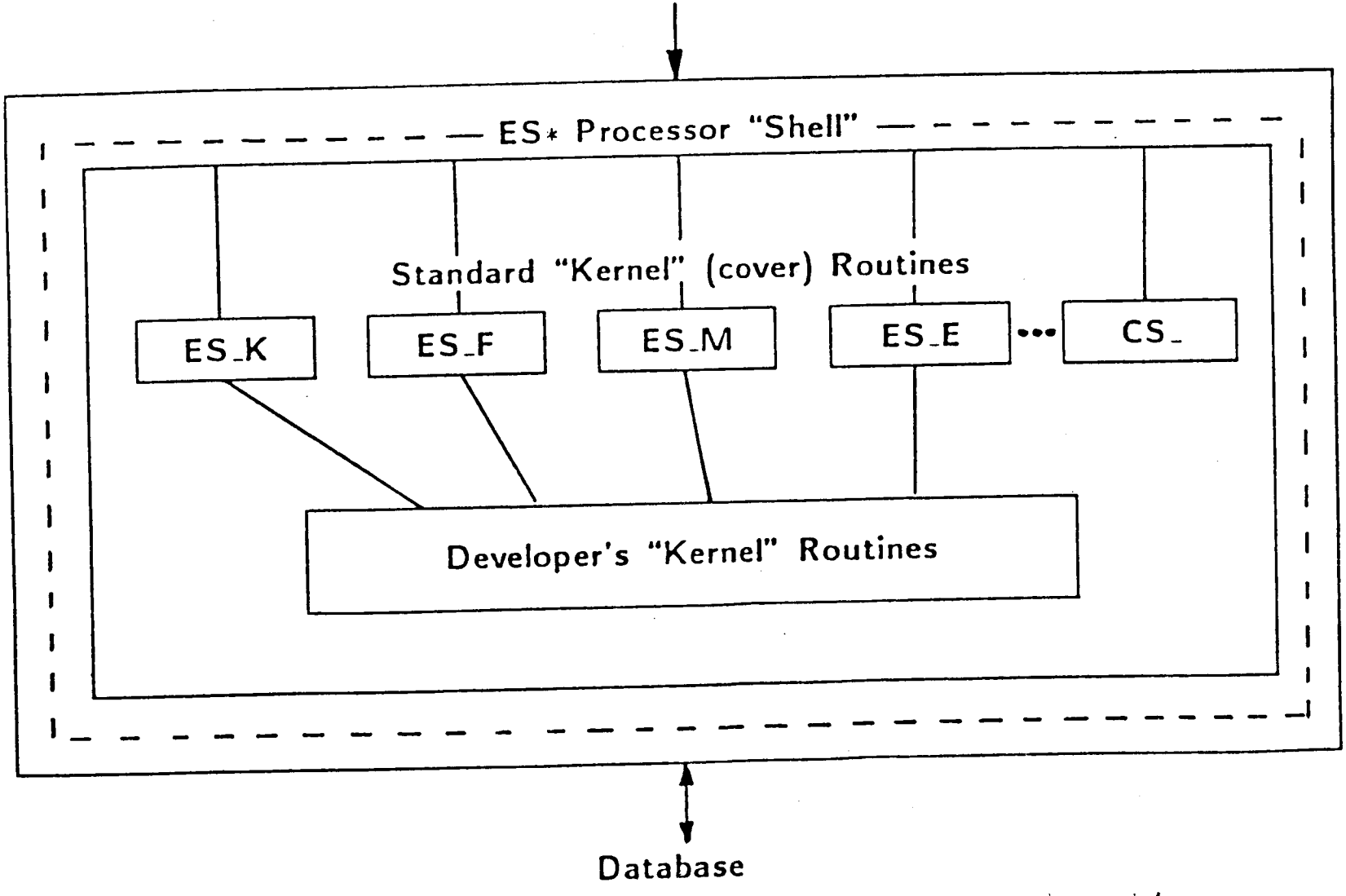
Another motive that spawned the development of a Generic Element Processor is our view that it should be easy to add new elements to the Testbed — for both linear and nonlinear analysis — so that the Testbed can provide a common "gauntlet" for new element technology, as well as serving as a strainer to filter out poor performers and identify good ones. In summary, elements should be easy to implement in the Testbed, to evaluate numerically, and to apply to complex problems (e.g., procedurized focal problems) without any substantial delays.

Generic Element Processor

APPROACH (visual)

Lockheed *CSM*

ES* Commands: FORM { FORCE | STIFFNESS | MASS ...}



GENERIC ELEMENT PROCESSOR
APPROACH

This chart gives a conceptual view of the Generic Element Processor. It features a standard outer software "shell" that processes user commands (such as FORM STIFFNESS) and handles all input/output from/to the global database (via GAL). Inside this shell is the non-standard "kernel" supplied by the element developer, which may be written in just about any style or granularity, provided that it is in FORTRAN (for now). Finally, a standard set of *shell/kernel* interface routines have been defined for each of the many element functions. The guts of these interface routines are provided by the element developer, while the subroutine names and argument lists are standardized. The function of the interface routines is to transition between the standard argument lists and the developers personal code in whatever manner is natural for the developer.

An additional function of the Processor "shell" is to provide automatic corotational updating capability for geometrically nonlinear (arbitrarily large rotation) analysis, so that element developers do not *have* to be concerned with such matters (however, they can override this option if they want to). Note that this automatic geometric nonlinearity feature has been implemented in a manner that is independent of the Testbed architecture (database and command language) — as an inner shell — to facilitate evolution of these two quite different functions.

GENERIC ELEMENT PROCESSOR

USER'S VIEW

Lockheed Palo Alto *CSM Testbed Development*

```
*call ES ( function = 'FORM { STIFFNESS | FORCE ...}'
          elt_dis_ds = TOT.DISP.<step>
          elt_rot_ds = TOT.ROTN.<step> ...)
```

```
*procedure ES ( function; elt_dis_ds; elt_rotn_ds ...)

    *do es_proc = 1, num_es_proc

      [xqt <es_proc_name>
        [function]

    *enddo

*end
```

GENERIC ELEMENT PROCESSOR
USER'S VIEW

The Generic Element Processor is not really a single Processor at all, but rather a standard Processor "shell" that can be used to construct an unlimited number of specific element Processors. The user (i.e., analyst or Procedure writer) interface to element Processors created from the Generic Element Processor mold can be either direct — through commands such as `FORM STIFFNESS`, `FORM FORCE`, etc., or through a high-level Procedure called `ES` (for *S*tructural *E*lements). The advantage of the Procedure interface is that it will run all element Processors that are active for the current problem, ***automatically***, via database lookups and internal `DO` loops. This saves the user the trouble of explicitly executing each element Processor for each function required. Note that the `ES` Procedure call has arguments that are analogous to the Processor commands. For example, `function = 'FORM STIFFNESS'`, defines the command to be executed by all structural element Processors, and `elt_dis_ds = TOT.DISP.<step>`, defines the name of the displacement dataset to be used by all structural element Processors.

GENERIC ELEMENT PROCESSOR

DEVELOPER'S VIEW

SHELL/KERNEL INTERFACE ROUTINES

ES_D	Parameter Definition
ES_I	Initialization
ES_KM	Material Stiffness
ES_KG	Geometric Stiffness
ES_FI	Internal Force
ES_FE	External Force
ES_MC	Consistent Mass
ES_MD	Diagonal Mass
ES_TR	Transformations
ES_N	Unit Normal Vectors

⋮

GENERIC ELEMENT PROCESSOR
DEVELOPER'S VIEW

For the element developer, of main interest is the set of shell/kernel interface routines that must be completed to bring a newly implemented element up to full functionality. A partial list of these routines is shown in this chart. (The list is subject to grow as research demands.)

While most of these routines correspond to Processor commands (which the element developer doesn't have to deal with), more than one shell/kernel interface routine may be invoked as a result of a single command. For example, the command `FORM FORCE/INT` may cause `ES_D` (element definition), `ES_E` (element strains) and `ES_FI` (element internal force vector) to be called by the generic Processor shell. Note also that the element developer does not currently have to provide constitutive routines. A standard set of *linear* constitutive routines is built in to the shell. Eventually, we expect this standard constitutive interface to evolve into a separate Generic Constitutive Processor for general linear and nonlinear material models (including stress/tangent-modulus calculation and failure analysis).

GENERIC ELEMENT PROCESSOR

DEVELOPER'S VIEW (continued

ES_D (eltnam, eltnum, ctls, DEFS, DOFS, NODES, PARS, Status)

ES_FI (eltnum, ctls, defs, dofs, nodes, pars, store, x, d, s, FI, Status)

DEFS(pdNEN) — number of element nodes
(pdNIP) — number of integration points
(pdCLAS) — idBEAM | idSHEL | idSOLI
(pdNDOF) — number of freedoms per node

(pdESKM) — ES_KM implementation status

:

GENERIC ELEMENT PROCESSOR

DEVELOPER's VIEW (continued)

This chart provides a closer look at the shell/kernel interface routines, showing sample argument lists and some of the important parameters contained in one of the argument arrays. Notice that we are compensating for FORTRAN as much as possible, by using *array* data structures (wherever appropriate) to facilitate later extensions.

Subroutine ES_D (element Definition) is different from the other interface routines. For it, the element name (eltnam) is input, and the basic element definition parameters (DEFS), a nodal freedom activity table (DOFS) and a node sequence array (NODES) are output. In contrast, action routines such as ES_FI (element internal force) employ DEFS, DOFS and NODES as input — in addition to a control array (CTLS) and a research parameters array (PARS). In this case, the element nodal coordinates (x), displacements (d) and stresses (s) are input, and the internal force vector (FI) and subroutine return status (Status) are output.

The DEFS array is an expanding list of parameters that enable the element developer to convey a precise description of the element to the generic Processor shell — for example to enable the shell to perform certain corotational functions. It also contains parameters such as ESKM, ESKG, ESFI, etc., which convey the implementation status of the corresponding element function. The entire DEFS array is stored in the global database for users (and developers) to query before, during or after an analysis.

GENERIC ELEMENT PROCESSOR

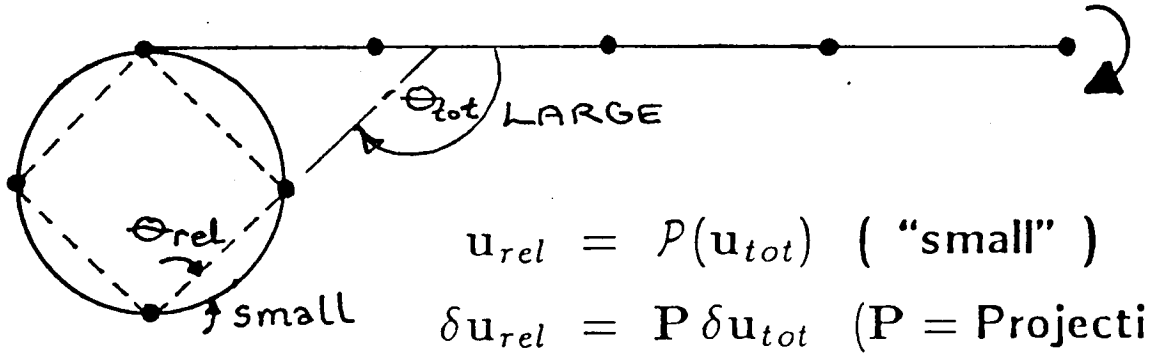
FEATURES

- MODULARITY
  - — Developers may work independently (insulated from arch.)
  - — All elt. fns under one roof (object-oriented)
- EXTENDIBILITY
  - — Number of elt. types/fns/parameters open-ended
  - — Number of elt. Processors open-ended (ES1, ES2, ...)
- SIMPLICITY
  - — Shell/kernel interface easy to build (repackage in days!)
  - — Procedure interface is high-level (e.g., FORM STIFFNESS)
- SELF-DESCRIPTION
  - — Standard interface makes elt. software accessible to others
  - — Database contains elt. parameters/status for user query
- AUTOMATIC GEOMETRIC NONLINEARITY (Corotational)

GENERIC ELEMENT PROCESSOR
FEATURES

In summary, the Generic Element Processor is modular, extendible, simple to use/develop, self-described and facilitates geometric nonlinearity. This last feature is made possible by implementation of a corotational utility library within the Processor shell. This corotational shell enables linear elements to be employed for problems involving arbitrarily large rotations — with no additional work by the developer beyond the usual linear subroutines (including one for the geometric stiffness matrix). Moreover, while the corotational option is the default, alternate methods for handling geometric nonlinearity are accommodated by providing the developer with the element displacement vector as input argument to most of the shell/kernel interface routines. Also, the special nodal rotation quantities that are automatically provided in large rotation analysis for beam and shell elements (i.e., rotation pseudovectors corresponding to orthogonal triads) can either be employed, ignored, or replaced, by the element developer who wishes to incorporate a completely different type of nonlinear formulation. Note that total Lagrange (TL), updated Lagrange (UL) and corotated Lagrange (CL) formulations all fall naturally within the present framework.

COROTATION
(In a Nutshell)



$$\mathbf{u}_{rel} = \mathcal{P}(\mathbf{u}_{tot}) \quad (\text{ "small" })$$

$$\delta\mathbf{u}_{rel} = \mathbf{P}\,\delta\mathbf{u}_{tot} \quad (\mathbf{P} = \text{Projection})$$

$$\mathbf{K}_{tot} = \mathbf{P}^T\mathbf{K}_{rel}\mathbf{P} + \mathbf{K}_{hot}$$

— Upgrades linear elements to geom. nonlinear (ELEMENT-INDEPENDENT)

— Projects out rigid-body errors (warping sensitivity)

— Adds h.o.t.s for Consistent Tangent Stiffness [Rankin/Nour-Omid]

— Uses consistent NODAL rotation update: $\mathbf{R}_{n+1} = \mathbf{e}^{\delta\theta}\,\mathbf{R}_n$

COROTATION
(In a Nutshell)

Lockheed Palo Alto *CSM Testbed Development*

Corotation is a method for making large rotations relative to an inertial frame *look* like small rotations at the element level. This is achieved by defining an element coordinate frame (for each element) that rotates with the element (as defined by its nodal coordinates). The rigid motion of this frame is then "subtracted" from the total motion of the nodes, leaving relative translations and rotations that can be made arbitrarily small by simply refining the mesh (i.e., by adding more element frames). Note that for the rotational degrees of freedom, "subtracting" the rigid rotations really amounts to multiplication of orthogonal matrices, and that the updating of these matrices (or triads) at nodes from step to step (before removing the rigid motion) must be done in a manner that preserves orthogonality.

Once the relative motions at nodes have been rendered sufficiently small, they may be used in simplified strain-displacement relations. For example, for shell elements, either linear strain-displacement relations or so-called moderate rotation nonlinear strain-displacement relations may be used in conjunction with the corotational procedure. The effect of using nonlinear versus linear element strain-displacement relations usually amounts to an increase in accuracy, which can alternatively be achieved by adding more elements.

Corotation is not a new idea (e.g., Belytchko [5], Wempner [6] and others were some of the innovators) but we have given it some new twists [7]. First, we have made it more *element-independent* by generalizing it to beam, shell and solid elements and providing generic software utilities for all three classes; and second, we have established a firm mathematical foundation by deriving it from variational principles and identifying its correspondence to a *projection* operator (P). By capitalizing on this projection operator idea, we have — through consistent linearization of the variational functional — come up with an element-independent higher-order stiffness matrix that ensures quadratic convergence in the context of a Newton-Raphson nonlinear solution procedure. A suprising fringe benefit is that the projection matrix, $\mathbf{P}$, that emerges from linearization, seems to correct rigid body errors (e.g., warping sensitivy) even for linear analysis! This may have a profound affect on various old and new element formulations (e.g., Szabo's p-refinement elements have implicit rigid body errors for low-order *p*olynomial approximations).

GENERIC ELEMENT PROCESSOR

PRIOR STATUS (August)

- Operational for Linear and Geom. Nonlinear Shell Statics
- Element Processors:
  - ES1 C^0 shell elements (4/9-ANS and 4/9/16-SRI)
  - ES2 C^1 shell elements (4/"9"-HYB)
- Analysis Procedures:
  - L_STATIC_1
  - L_STABIL_0 / L_STABIL_1
  - NL_STATIC_1 [Crisfield/Riks +]
- Applications
  - "Patch" Tests
  - Stiffened Shell Buckling
  - Simple Shell Post-Buckling

GENERIC ELEMENT PROCESSOR

PRIOR STATUS (August)

Lockheed Palo Alto

As of last August, we had used the generic element Processor "shell" to construct two specific element Processors, namely, ES1, which contains a family of C^0 shell elements, and ES2, which contains a family of C^1 shell elements. The C^0 (transverse shear deformable) shell elements in ES1 include 4- and 9-node assumed natural-coordinate strain (ANS) elements by Park and Stanley [8], and various selective/reduced integrated (SRI) elements (as in [9]). The C^1 (transverse shear rigid) shell elements in ES2 include flat and curved (exact-geometry) 4-node hybrid (HYB) elements by Kang and Pian [10]. Both the ANS and HYB elements in the two Processors, respectively, are formulated in terms of a local natural-coordinate basis, employing covariant strain components. Hence mesh distortion sensitivity is reduced considerably with respect to elements that employ local-Cartesian strain components for discretization. Collectively, Processors ES1 and ES2 represent (we believe) the state-of-the art in shell element technology.

We had also developed Procedures for linear and nonlinear static analysis, and had applied Processors ES1 and ES2 (via these Procedures) to simple linear and nonlinear test cases.

GENERIC ELEMENT PROCESSOR

Δ-PROGRESS (Aug–Nov)

Lockheed Palo Alto *CSM Testbed Development*

- Cleanup/Testing (Iteration)
- New Element Processors:
  - — ES3 Modified SPAR Hybrid Solid elts [Aminpour]
  - — ES4 " " " Shell " "
  - — ES5 STAGS 410 Shell elt [Brogan]–NAVY
  - — ES6 2/3-Node Hybrid Beam (Truss/Stiffener) [Kang]
  - — ES6 Improved ANS Shell [Park/Stanley/Cabiness] –NAVY
- New Analysis Procedures:
  - — NL_STATIC_1 extended to Specified Displacements
- New Applications:
  - — Postbuckling of Composite Stiffened Panels

GENERIC ELEMENT PROCESSOR

Δ-PROGRESS (Aug–Nov)

Lockheed Palo Alto

CSM Testbed Development

Between August and November, we have performed several iterations on the generic element Processor "shell", making it easier for both element developers and Procedure writers to interface with, and adding some new pre- and post-processing capabilities as well (e.g., stress transformation/archival and automatic freedom supression commands).

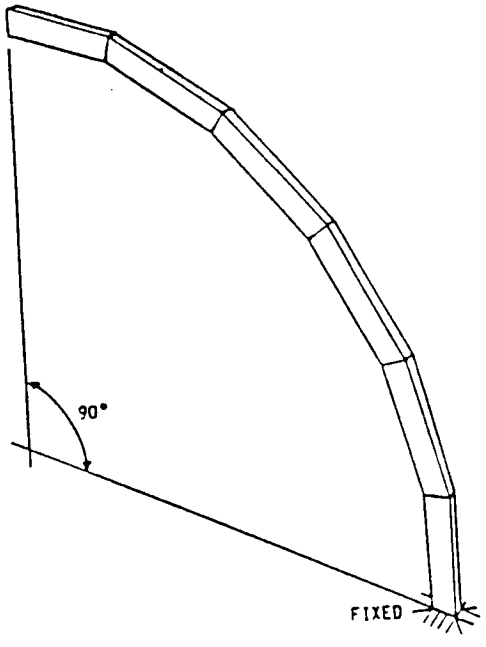
New element Processors, constructed with the generic element Processor shell, were implemented both at LPARL and at LARC. Dr. Mohammad Aminpour (LARC) re-packaged a set of improved SPAR hybrid shell and solid elements within Processors ES4 and ES3, respectively, following a one-week visit by Dr. Aminpour to LPARL. Subsequently, we re-packaged the workhorse shell element used in the STAGS code [11] (therein called the "410" element) as Processor ES5.* We also made a preliminary implementation of a set of straight and curved C^1 hybrid beam elements within Processor ES6; these elements are compatible with the shell elements implemented in Processor ES2 by Dr. David Kang.

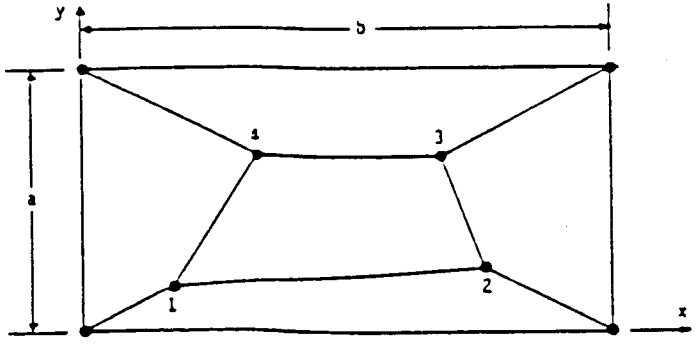
Finally, we extended the nonlinear static analysis Procedure, NL_STATIC_1, based on Crisfield's arclength method [12], to specified displacement loading, and used it to analyze the postbuckling response of I-stiffened composite panels. The finite element models used ranged from 4000 to 9000 degrees of freedom.

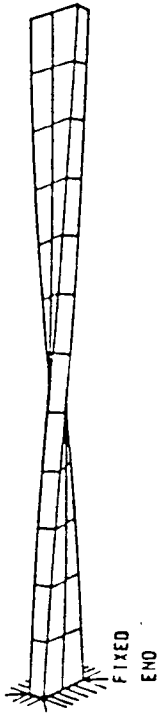
* The NAVY (NSRDC) has also contributed some funding towards this effort.

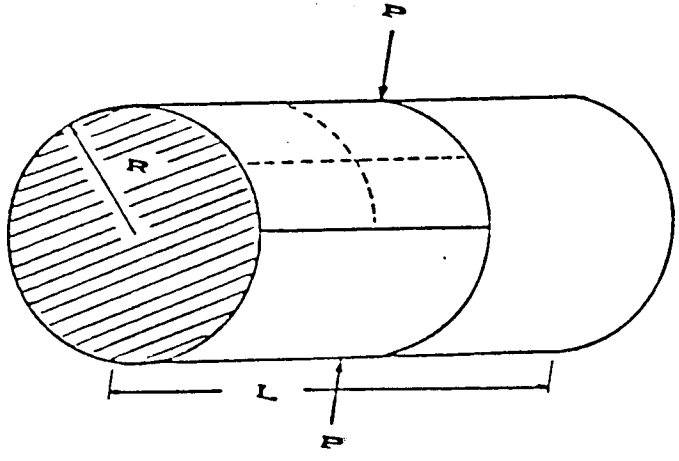
ELEMENT TEST CASES

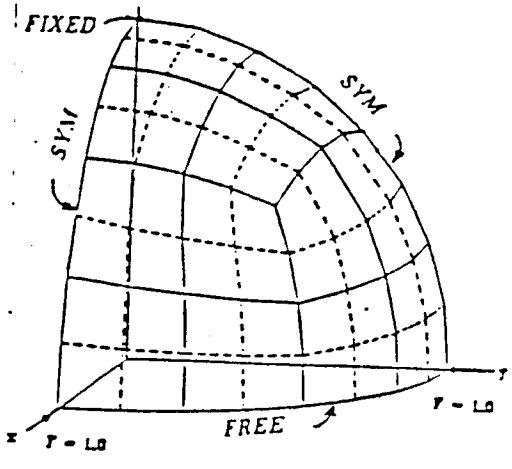
LINEAR











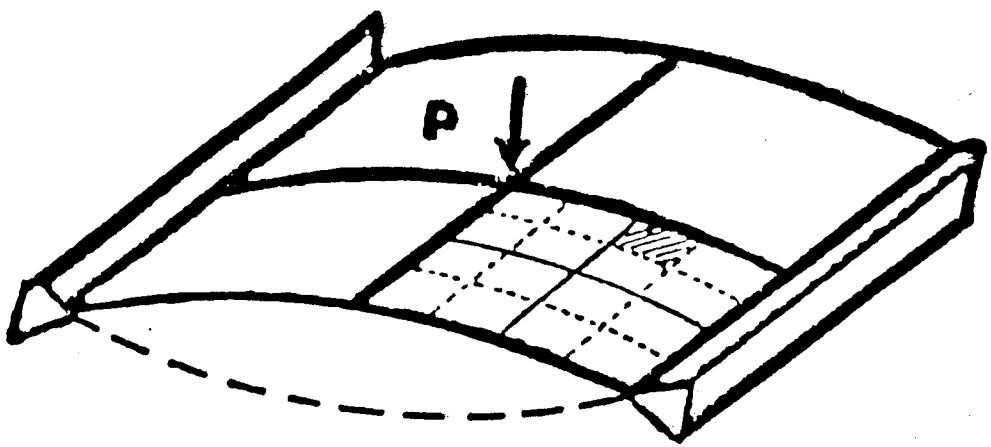
ELEMENT TEST CASES

LINEAR

Before applying a new element or solution algorithm to a complex problem, it is wise to first verify/evaluate it using standard benchmark problems. Shown in this chart are some of the linear test cases proposed by MacNeal/Harder [13]. They represent a series of patch tests and covergence tests that tend to bring out element pathologies. For example, the pinched hemisphere (bottom right) requires shell elements to display nearly inextensional bending behavior in locally warped configurations (quadrilateral element corner points do not all lie in the same plane). We have also added some of our own cases to this gauntlet, including gradually distorted mesh versions of the pinched cylinder problem. We have found that, typically, elements that can handle the pinched hemisphere and distorted-mesh pinched cylinder problems well tend to be good candidates for production applications.

ELEMENT TEST CASES

NONLINEAR: Hinged Cylinder



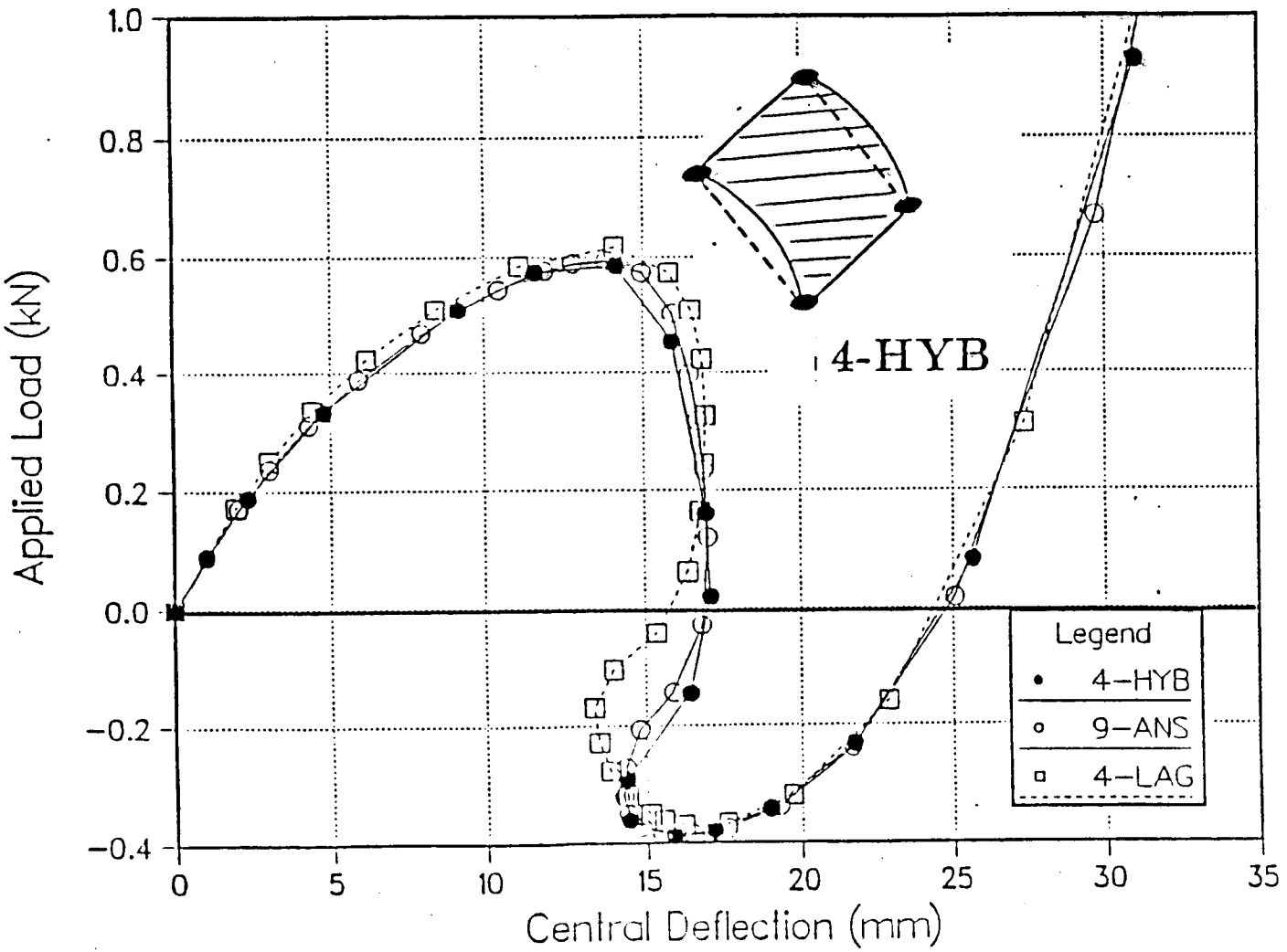
NONLINEAR: Hinged Cylinder

The linear test cases are necessary but not sufficient for certifying a new element. Shown in this chart is a deceptively simple nonlinear test case that has become a standard: the hinged cylinder. While only a small number of plate/shell elements should be necessary to model one quadrant of the shell, the geometrically nonlinear snap-through behavior requires a sophisticated solution algorithm, and checks out the element corotational interface as well.

HINGED CYLINDER RESULTS

C^0 vs C^1 Elements

Lockheed Palo Alto *CSM Testbed Development*



HINGED CYLINDER RESULTS

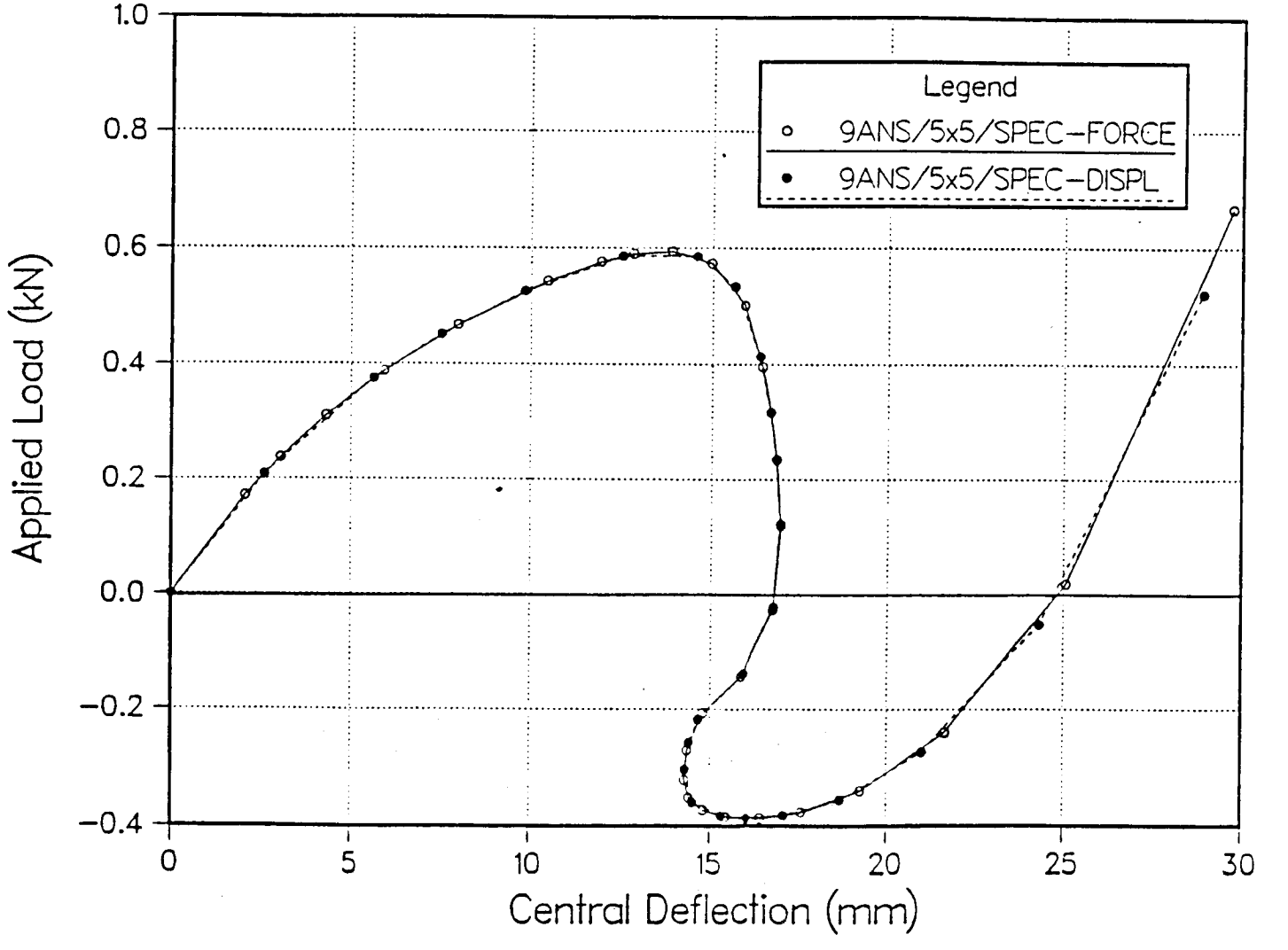
C^0 vs C^1 Elements

One of the things the hinged cylinder problem was used to check was the automatic corotational option built in to the generic element Processor. This chart shows the load versus displacement (under the load) static response for several different elements: the 4-node Lagrange element (4-LAG) and 9-node ANS (9-ANS) elements of Processor ES1, and the "curved" 4-node hybrid element (4-HYB) of Processor ES2. Note that the 4-LAG element is the least accurate (as expected), and while the 9-ANS and 4-HYB yield practically identical curves, the latter element required only about half as many load steps to traverse the curve. This is indicative of the power of the curved 4-HYB element, which emanates from its use of a cubic, C^1 continuous displacement field and parabolic (9-node based) geometry. Note that the comparison is for the same number of *nodes* (not elements), so that there are actually four times as many 4-HYB elements as 9-ANS elements in the meshes used. (Also, there is a flat version of the 4-HYB element, not shown, which did not perform nearly as well as the curved version.)

HINGED CYLINDER RESULTS

Specified Disp. vs Force

Lockheed Palo Alto *CSM Testbed Development*



HINGED CYLINDER RESULTS

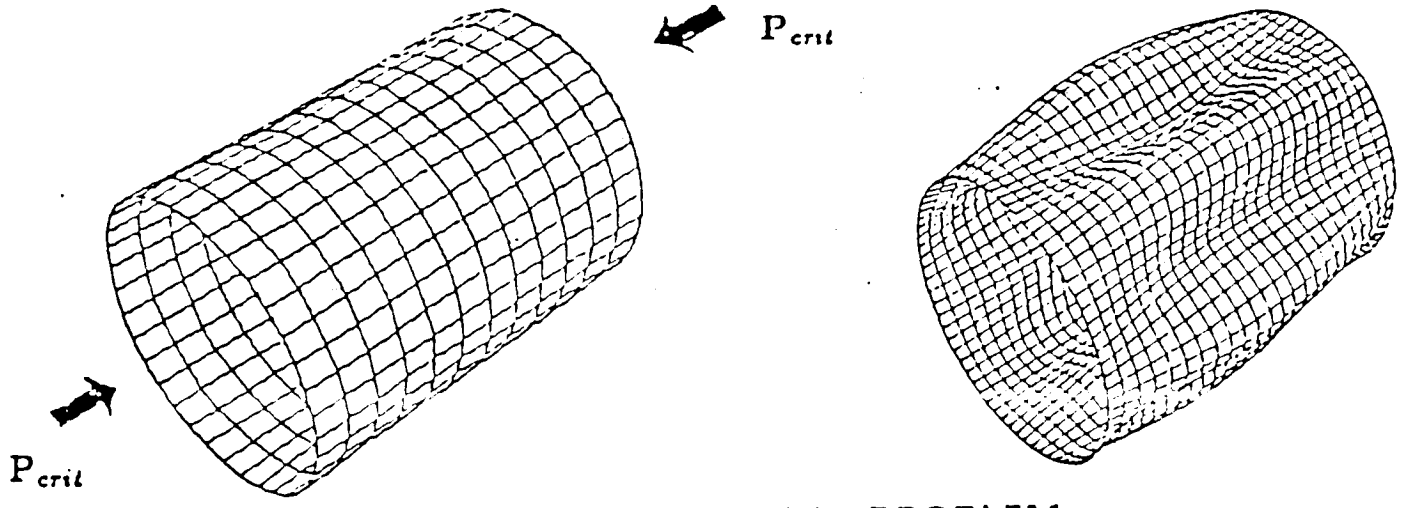
Specified Disp. vs Force

The second thing checked by the hinged cylinder problem was the newly added specified displacement loading capability to Procedure NL STATIC_1. Shown in this chart are the central force vs displacement curves for both specified force loading and specified displacement loading. Note that there is relatively little difference between the curves or the number of load steps required. However, the load steps are different, due to differences in the scaling of the arclength constraint equation and in the error norms used for the two cases.

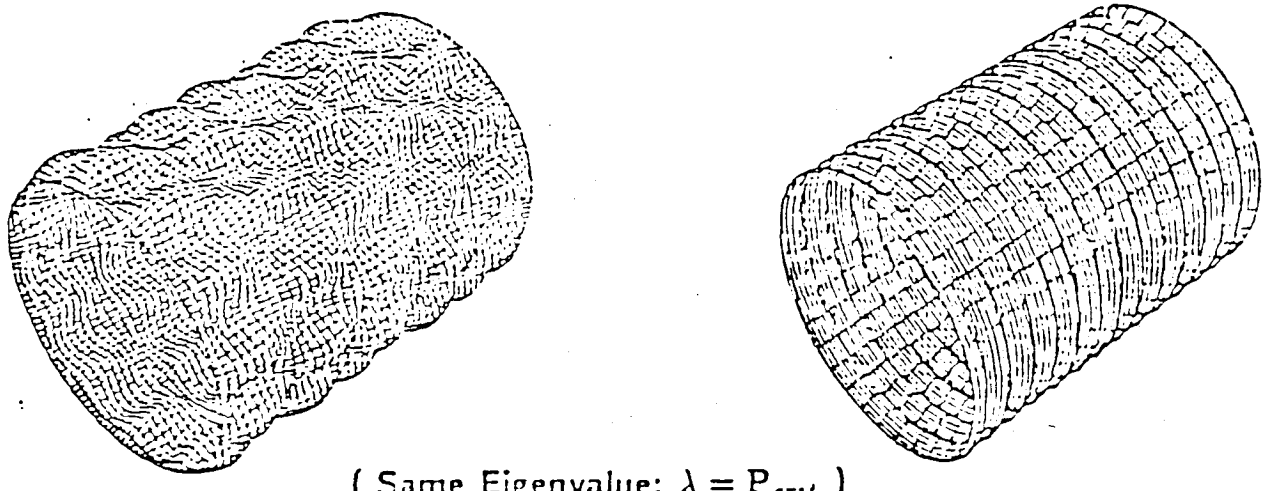
ELEMENT TEST CASES

NONLINEAR: Compressed Cylinder

Lockheed Palo Alto *CSM Testbed Development*



CLASSICAL BUCKLING PROBLEM



(Same Eigenvalue: $\lambda = P_{crit}$)

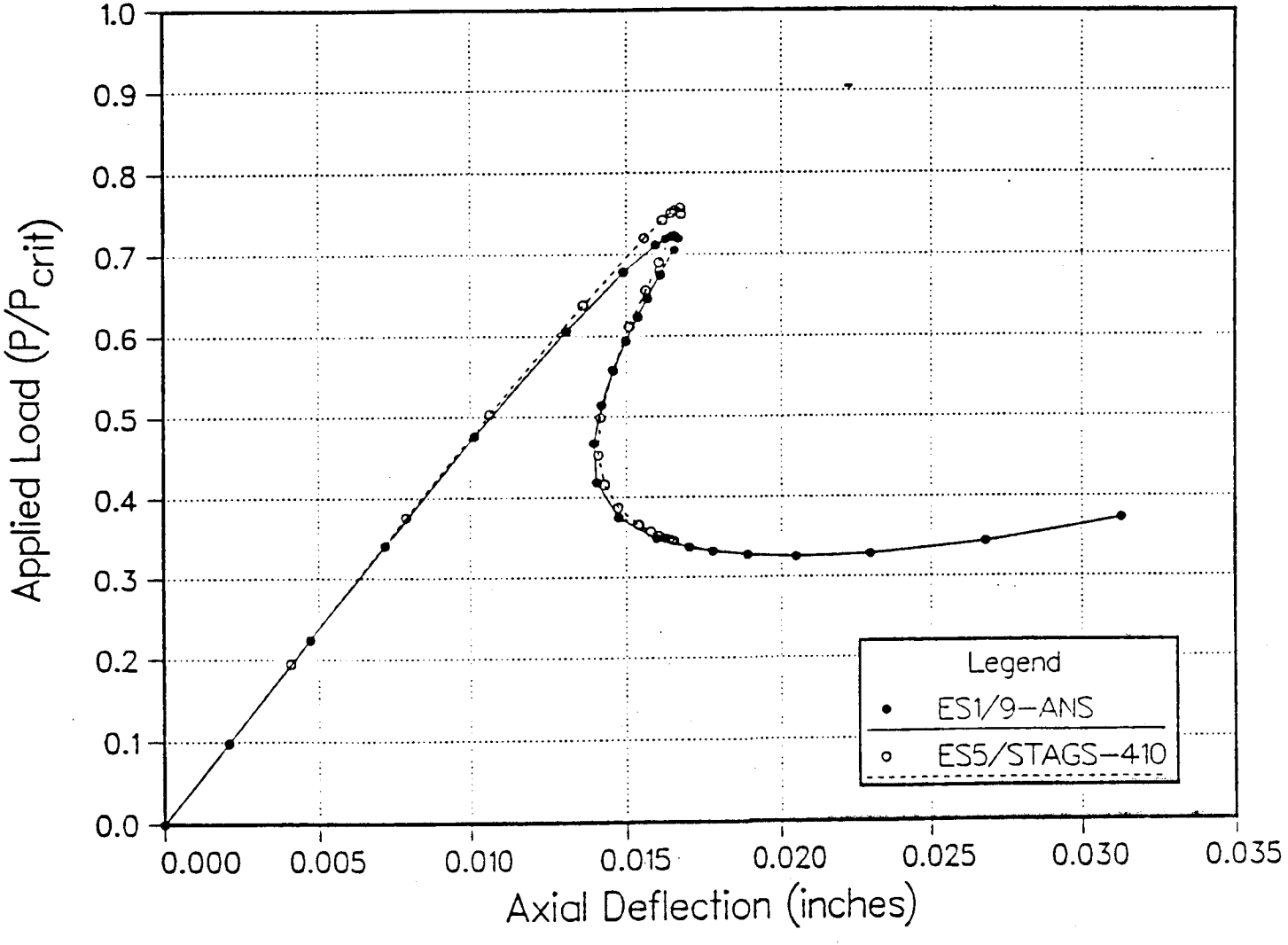
NONLINEAR: Compressed Cylinder

As a final pre-requisite for the solution of the I-stiffened panel postbuckling problem, we revisited the classical compressed cylinder postbuckling problem, shown in this chart. One noteworthy feature of this (also deceptively simple) problem is that there are many buckling modes (eigenvectors) at nearly the same buckling load level (eigenvalue). Another important, and related, feature is the sensitivity of thin cylindrical shells under axial loading to geometric imperfections. In the analysis, we imposed an imperfection corresponding to the diamond-pattern buckling mode shown in the lower left of the chart, using a maximum radial amplitude equal to one-tenth of the shell thickness (radius/thickness = 300).

COMPRESSED CYLINDER RESULTS

9ANS vs E410

Lockheed Palo Alto *CSM Testbed Development*



COMPRESSED CYLINDER RESULTS

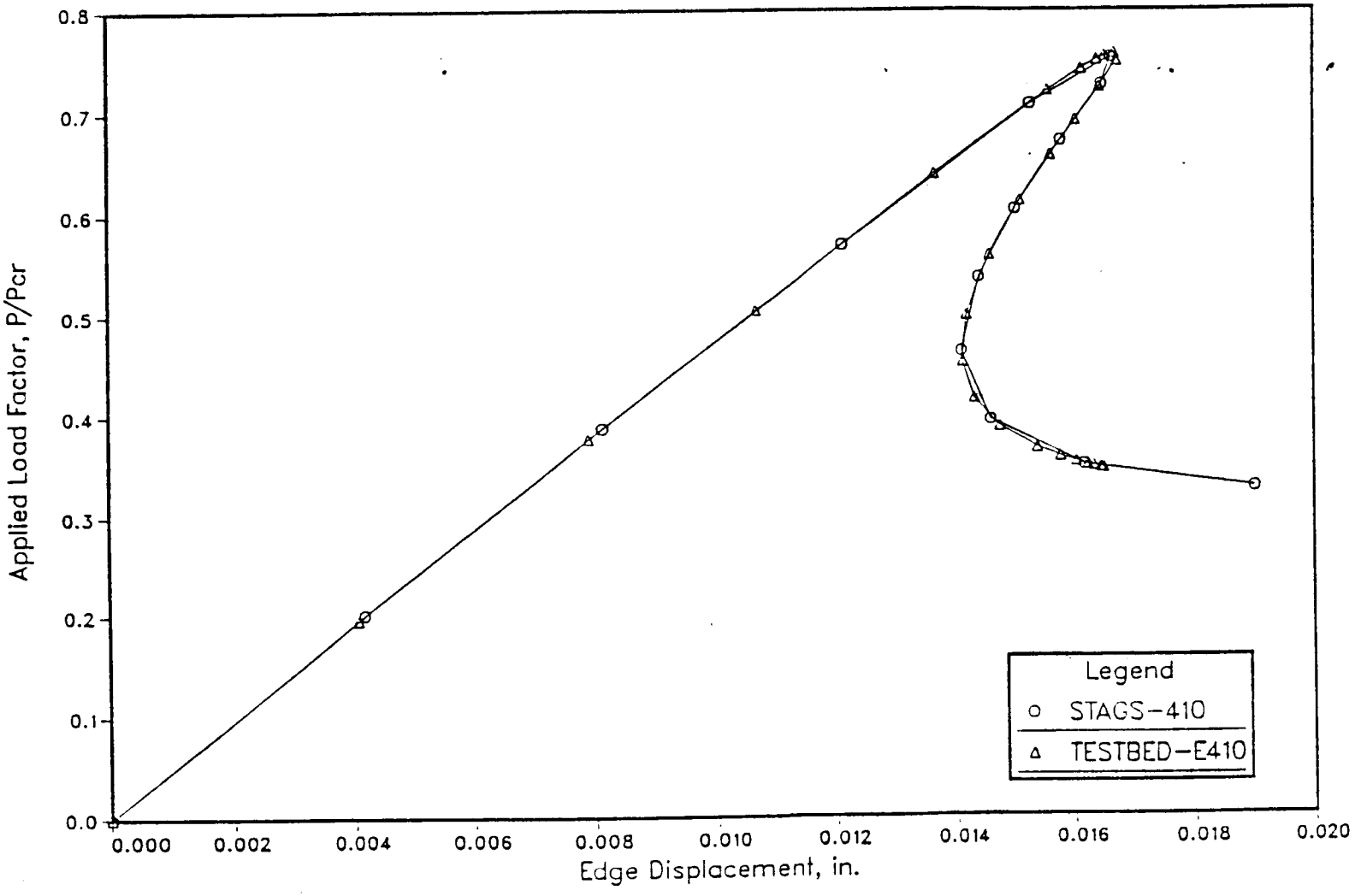
9-ANS vs E410

We used this problem primarily to verify the new implementation of the STAGS 410 element within Testbed Processor ES5 (as element type E410). However, we had some difficulties. The present chart shows a plot of normalized axial load versus end-shortening for both the 9-ANS shell element and the new E410 shell element (in Processors ES1 and ES5, respectively). Note that the E410 element has a slightly higher peak load than the 9-ANS element, but the two curves align closely thereafter. That is until the curve begins to flatten out again. At this point (for an axial deflection of about .017) the E410 element begins having convergence difficulties and eventually fails to converge for any step size. This behavior was quite baffling at first, so we proceeded to perform the same analysis with the STAGS code (see next chart).

COMPRESSED CYLINDER RESULTS

STAGS vs CSM (E410)

Lockheed Palo Alto *CSM Testbed Development*



COMPRESSED CYLINDER RESULTS

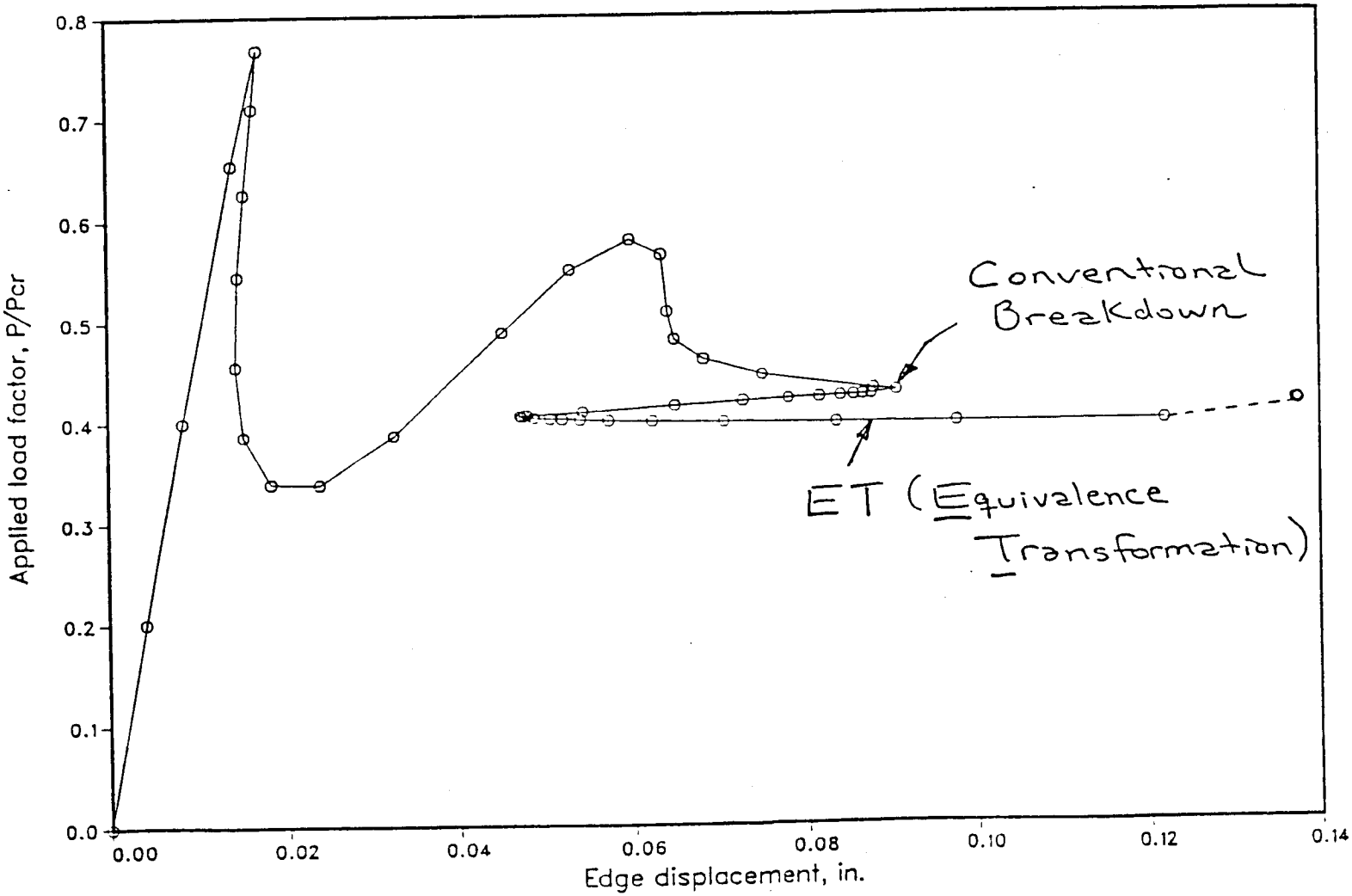
STAGS vs CSM (E410)

In this chart we compare load/displacement curves for the E410 element with both the CSM Testbed (Processor ES5) and with STAGS. That is, the same element is being employed in two different software systems, in order to verify the new Testbed implementation. We found that not only do the curves coalesce up to the point at which the Testbed implementation has convergence difficulty, but the element arrays generated by the Testbed were identical with those generated by STAGS for a given displacement configuration (we verified this independently by specifying an arbitrary nonlinear displacement configuration as input to the two systems). Thus, we have surmised that the convergence difficulties are probably due to hidden *single precision* operations being performed (on our 32-bit word VAX computers) in one or more of the SPAR matrix Processors. For example, accumulation of round-off errors could easily ruin the integrity of the factored stiffness matrix, even if the element stiffness matrices are maintained in double precision. We intend to verify this hypothesis as soon as the software is operational on the CRAY-2 computer at NASA/Ames. (The reader may be wondering why this "precision" problem didn't afflict the 9-ANS element. We can only postulate at the moment that it is due to the generally improved conditioning of C^0 element stiffness matrices over C^1 element matrices.)

COMPRESSED CYLINDER RESULTS

STAGS ET Breakthrough

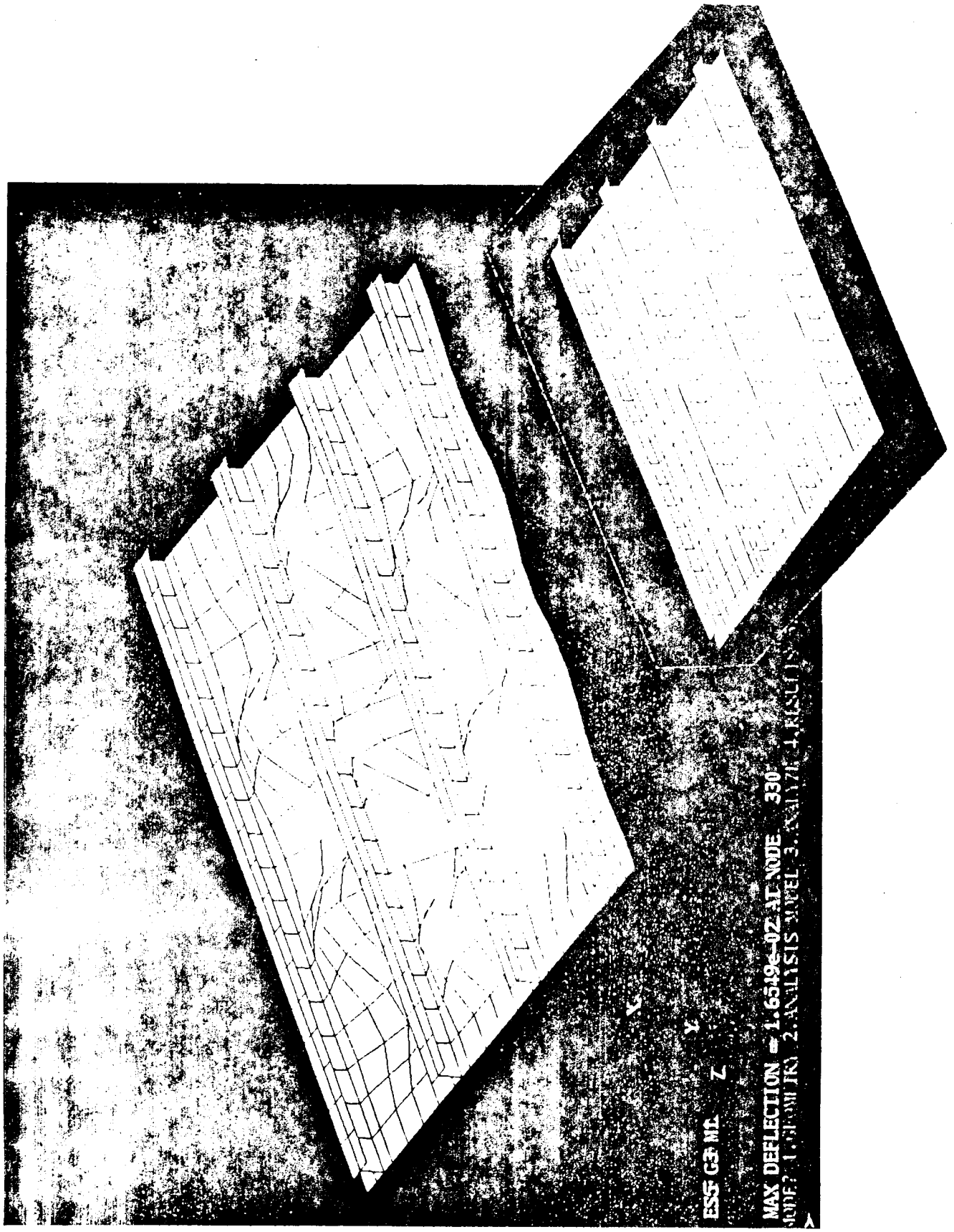
Lockheed Palo Alto *CSM Testbed Development*



COMPRESSED CYLINDER RESULTS

STAGS ET Breakthrough

The last curve to be shown here for the compressed cylinder has nothing to do, per say, with the Testbed. It shows how a new algorithm, called ET (Equivalence Transformation) [14], recently implemented in the STAGS code, allows the static solution to proceed beyond the point at which conventional, arclength algorithms break down (for all elements). With the ET algorithm, eigenvalue analyses are performed about these barrier points (i.e., using a nonlinear prestress state), and the lowest eigenvectors are used to temporarily reduce the system. The reduced system is then solved nonlinearly, locally, just to determine the direction of a stable equilibrium path beyond the barrier point. The reduced system is then expanded back to the full system, and the solution continued on the newly found path. Such techniques allow the user to explore solutions never before attainable (statically): perhaps too many solutions, since the "correct" one may be very hard to deduce. Nevertheless, we have plans to implement ET within the Testbed next year, as a high-level command Procedure, to make it more accessible to others.



ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

I-STIFFENED PANEL

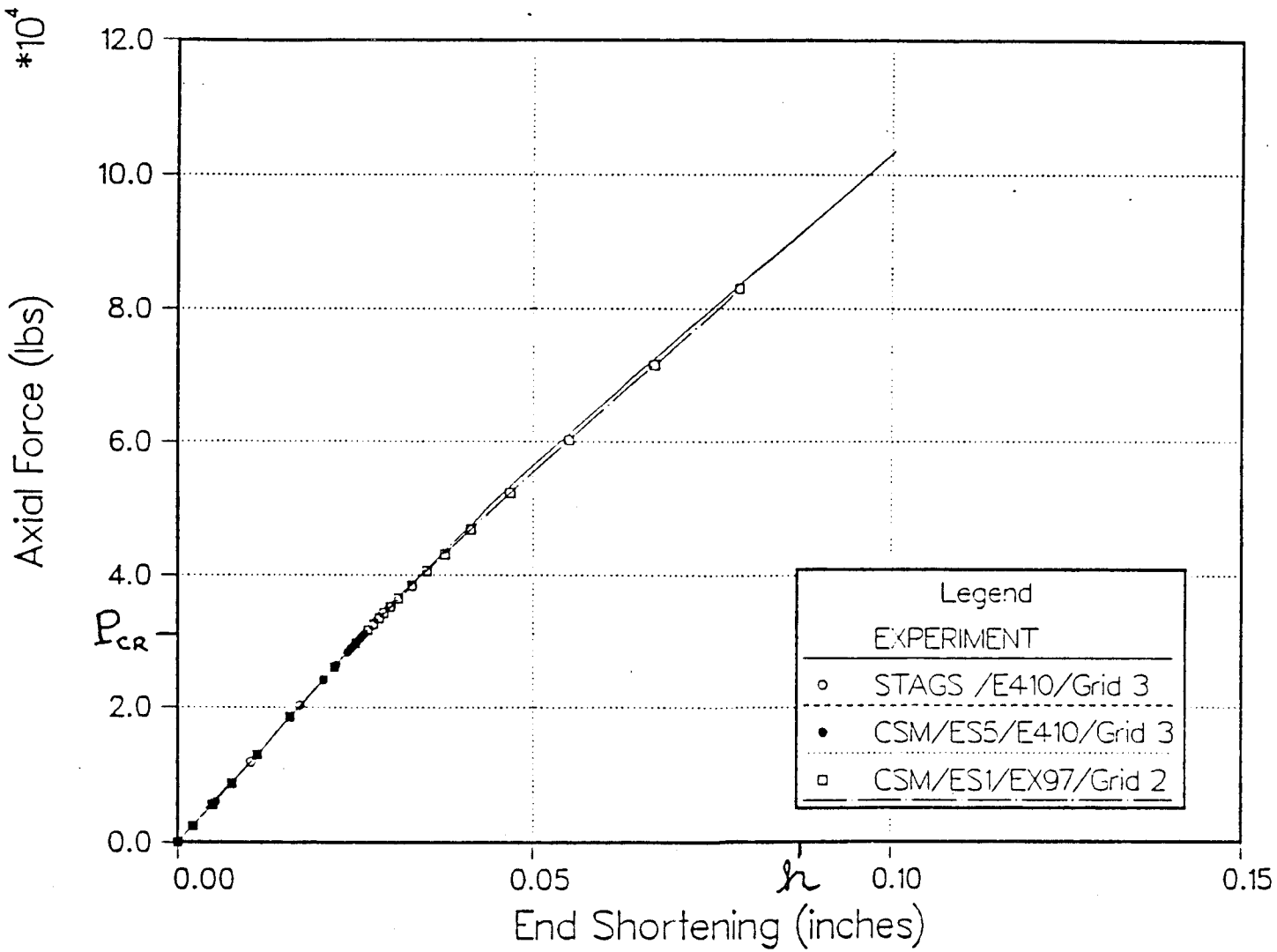
Buckling Results

At last, we come to the focal problem of the last three months: compression of a flat composite rectangular panel with four, equally spaced I-section stiffeners parallel to the long dimension. Analysis of this generic aircraft panel was proposed by NASA as a means for both verifying and validating the advanced shell postbuckling capabilities now in the Testbed. The specific choice corresponds to a specimen studied earlier by Drs. Jim Starnes and Norm Knight (NASA/LaRC), and published in the AIAA Journal [15]. Both experimental and analytical (STAGS) results were reported therein and hence provide a good basis for comparison here. Shown in the present chart is the initial geometry of the panel (inset) and the first buckling mode (eigenvector) corresponding to loading by uniform end-shortening. Note that there are 5 half-waves in the longitudinal direction, and 1 half-wave between stiffeners for this mode. The second buckling mode (not shown) differed by about 10% in value, and featured 6 longitudinal half-waves also with 1 half-wave between stiffeners. A small linear combination of these first two buckling modes (eigenvectors) was employed as a geometric imperfection for the nonlinear postbuckling analysis: specifically, 1 percent and .1 percent of the skin thickness were used, respectively, as the imperfection amplitudes for the two modes.

I-STIFFENED PANEL RESULTS

Load vs End-Shortening

Lockheed Palo Alto *CSM Testbed Development*



I-STIFFENED PANEL

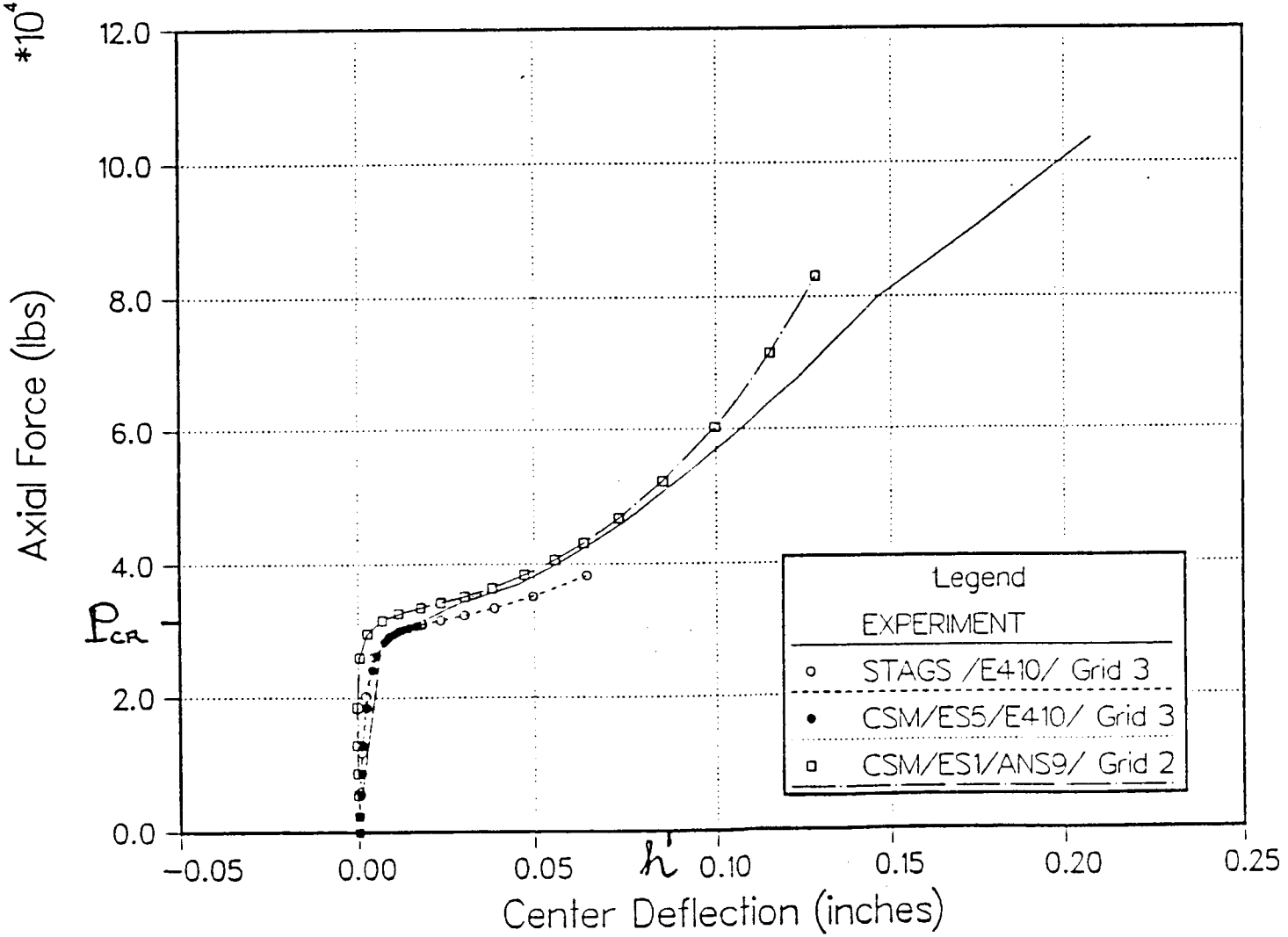
Load vs End-Shortening

Shown in this chart is the resultant axial force versus axial displacement (end-shortening) for the nonlinear response of the I-stiffened panel. Four curves are plotted, all of which are practically identical: experimental (solid line), a 600 element STAGS model using the 410 (ES5) shell element (open circles), a 600 element Testbed model using 410 shell element (solid circles), and a 400 element Testbed model model using the 9-ANS (ES1) shell element. Note that there is a slight "knee" in the curve just after the critical buckling load is attained. However, the full nonlinearity of the problem can only be appreciated by observing the lateral displacement component as a function of applied load (see next chart).

I-STIFFENED PANEL RESULTS

Load vs Center Deflection

Lockheed Palo Alto *CSM Testbed Development*



I-STIFFENED PANEL

Load vs Center-Deflection

In this chart the lateral displacement at the center of the I-stiffened panel is plotted versus the applied axial force, for the same cases as shown in the previous chart. Here, we can see the pronounced nonlinearity (i.e., "knee") as the load is increased beyond the critical buckling value. We can also see that, as before (see the Compressed Cylinder results), while the STAGS solution follows the experiment all the way to collapse, the same element in the Testbed (ES5/E410) has unresolvable convergence difficulties as it rounds the bend. Once again, we tentatively attribute this phenomenon to hidden single precision matrix operations.

Notice also that while the Testbed solution for the 9-ANS element (ES1/ANS9) does not have convergence difficulties, it begins to diverge from the experimental curve at about half of the collapse load. There are a couple of possible explanations for this. First, the 9-ANS element model is coarser than the 410 element model — there are fewer elements and less interpolating power — even though it involves more degrees of freedom (due to the unavoidable extraneous nodes in the stiffener cross-sections). Thus, refinement of the 9-ANS model may reduce the discrepancy with experiment. Second, due to differences in the accuracy of the eigensolutions for the 410 and 9-ANS models, the imperfections are not *exactly* the same; small differences in the imperfections can produce large differences in response for such imperfection-sensitive structures. Finally, note that the imperfections used were not experimentally measured, but rather "pulled out of the air", and may just happen to produce more accurate results for the 410 model than for the 9-ANS model.

These and other issues are now being studied, in addition to another, related focal problem, which features a *curved* panel, and for which experimentally measured imperfections are provided.

PITFALLS

- Mixed Precision
- DOF Renumbering Req's for Sparse Solvers ($10^7 \rightarrow 10^6$ storage)
- Specified Displacements and Arc-Length Methods
- Data Structures
  - — Overwriting essential for Nonlinear Analysis
  - — Should make more nominal — eventually

PITFALLS

Some important lessons were learned during the course of analyzing the preceding problems. The following is a partial list:

i) One should not risk using single precision for any CSM calculations on 32-bit word computers (such as VAXes and SUNs);

ii) Degree-of-freedom renumbering should always be performed in conjunction with sparse-matrix equation solvers, such as those currently used in the Testbed. For example, even though the initial node numbers for the I-stiffened panel models would have been nearly optimal for band and profile solvers, renumbering for *fill minimization* (via the Nested Dissection method) resulted in an order-of-magnitude decrease in required storage space for the factored matrix, and a similar reduction for factorization and solution times;

iii) Specified displacement loading must be added with care to arclength-type nonlinear static solution algorithms (more about this in the next chart);

iv) The current Testbed global data structures need revision. First, most datasets should use smaller granularity named (nominal) records so that the GAL write-in-place option can always be used; this is especially important for nonlinear analysis since large quantities of data are being written and re-written during the iteration process. Second, more *relational* data structurs should be adopted to facilitate pre- and post-processing, either interactively or via interface programs. In spite of these recommendations, we think that the transition to improved data structures should occur slowly, to permit the CSM user/developer community to adapt.

SPECIFIED DISPLACEMENTS AND

Arc-Length Methods

[Crisfield]

$$\|\Delta \mathrm{d}\|^2 + s\,\Delta\lambda^2\,\|\mathrm{f}_0\|^2 = \Delta L^2$$

$$\Delta \mathrm{d} = \begin{Bmatrix} \Delta \mathrm{d}^u \\ \Delta \mathrm{d}^s \end{Bmatrix} \Longrightarrow$$

$$\boxed{\|\Delta \mathrm{d}^u\|^2 + \Delta\lambda^2\,\|\mathrm{d}_0^s\|^2 = \Delta L^2}$$

SPECIFIED DISPLACEMENTS AND Arc-Length Methods

This chart highlights one of the pitfalls mentioned on the previous chart: introducing specified displacement loading into arclength-type solution algorithms, such as Crisfield's algorithm [12]. While Crisfield's arclength constraint equation omits the specified force ($\mathbf{f}_0$) loading term — to avoid scaling (s) problems — the analogous term for specified displacement loading must be included. This term can be turned on automatically by including the specified components of incremental displacement, $\Delta\mathbf{d}^s$, when taking the displacement norm, $\|\Delta\mathbf{d}\|$. Note that these components should be left out of other incremental displacement vectors (for example, when computing error norms), since the specified displacement components are, by definition, not considered as unknowns.

We have found — by experience — that if the $\Delta\lambda^2\|\mathbf{d}_0^s\|^2$ term is not included in the arclength (ΔL) constraint equation (last equation in the present chart), it is often not possible to drive the solution over limit points. This of course defeats the main purpose for using an arclength method in the first place. Note that such subtle points are not easily found in the literature.

CONCLUSIONS

- CSM Testbed Increasing our Productivity:
  - — Teamwork
  - — Interaction
  - — Useful Results
- Computational Efficiency not a Serious Drawback
- Generic Element Processor Proving Successful at LPARL
- Work needed in 3 other areas ...

CONCLUSIONS

As a result of our increasing use and development of the CSM Testbed, our own research laboratory is experiencing an increase in productivity as well. For the first time, we are able to work together via a common software interface, to interact on diverse tasks (when appropriate), and to produce results that are of visible use to others.

Furthermore, one of our fears regarding the Testbed has been alleviated: computational efficiency. Comparisions with the STAGS code yield run times within ±50% (depending on function), for problem sizes on the order of 5000 degrees-of-freedom. More statistics are now being gathered for benchmark documentation.

Regarding the generic element Processor, it does seem to expedite the introduction of new elements into the Testbed; for example the STAGS 410 element was introduced in about a week (once the necessary strings were cut from the STAGS code). Even less time for new-element entry is anticipated now that some of the bugs have been removed, and tutorial documentation (in preparation) should facilitate the process at remote sites.

Nevertheless, more work is needed both on the generic element Processor and in three other areas, as discussed in the following two charts.

PLANS

Generic Element Processor

Lockheed Palo Alto *CSM Testbed Development*

- Documentation
- Advanced Arc-Length Algorithm [Riks] – NAVY
- More on Beam and Solid Elements

- Test Dynamics
- Constraint (e.g., Contact) Elements
- Boundary Elements
- Non-Structural Elements: ET* EF*
- ERROR CONTROL
- Vectorization
- PARALLELIZATION: ES1 ES1 ES2 ...

PLANS

Generic Element Processor

Lockheed Palo Alto *CSM Testbed Development*

Development of the Generic Element Processor should be an ongoing process, with extensions added to accommodate the items mentioned in the present chart. Of particular importance, is the introduction of error control features, which will commence with the implementation of a family of p-refinement elements supplied by Barna Szabo. Such extensions, however, will not be isolated within the element Processor; for example, new high-level *Procedures* will have to be developed for adaptive refinement; and to exploit the p-refinement technique, new matrix utilities will be required as well (see next chart).

Another major thrust will be in the area of parallel processing, where independent element Processors will be mapped to hardware processors, probably according to a substructure partitioning scheme such as that proposed by Nour-Omid and Ortiz [16] in their "cut and paste" solution algorithm. Note that internal *vectorization* of element Processors is also on the agenda, having formulated a tentative plan for providing element kernels with "blocks" of element data instead of one element at a time. This would also provide a mechanism for automatic vectorization of element developer's code, should the need arise.

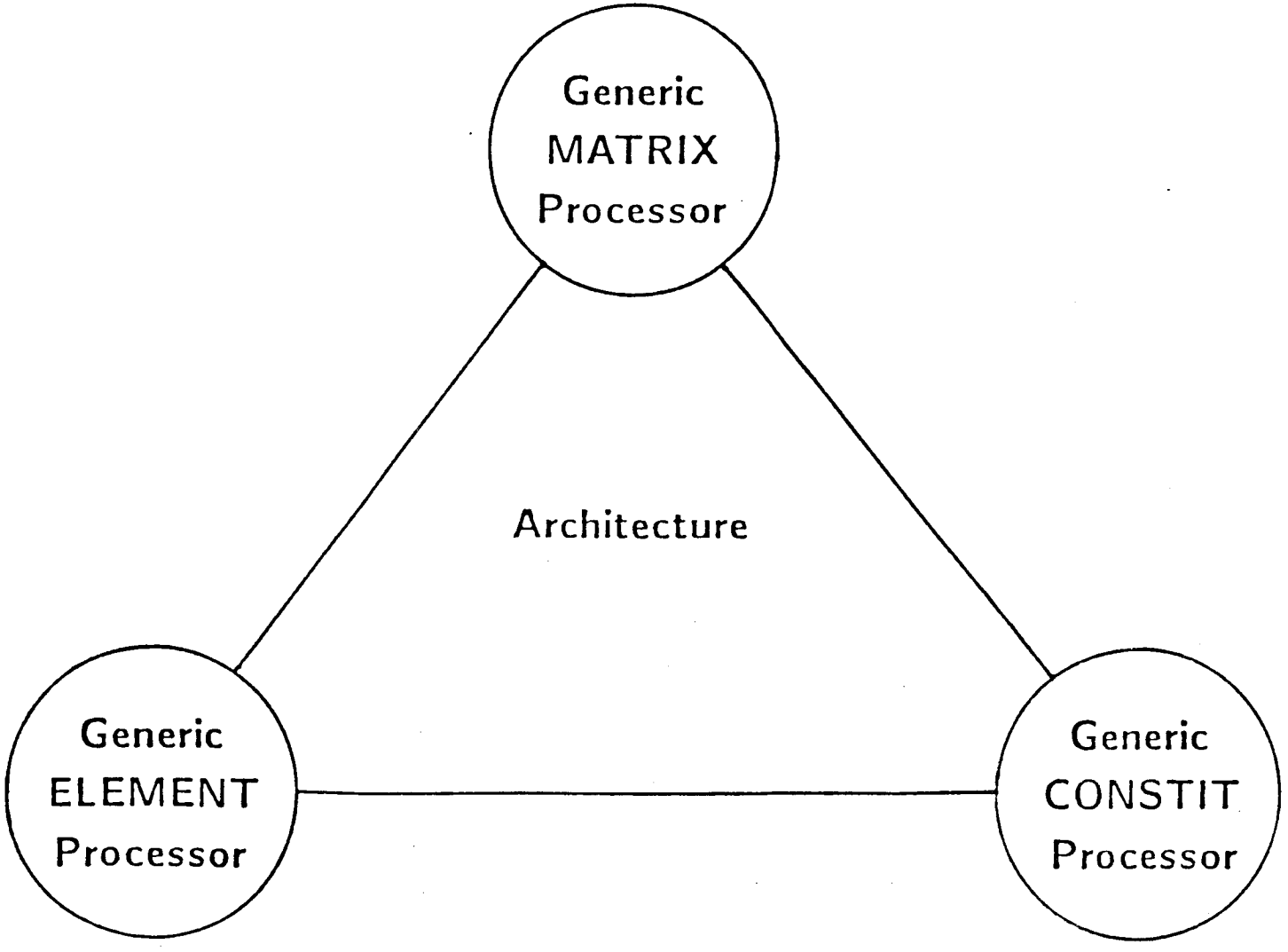
Finally, the present generic (structural) finite element Processor "shell" concept could easily be extended to non-structural elements.* For example, Generic Fluid Element Processors (EF*) and Generic Thermal Element Processors (ET*) would be useful in constructing procedures for structure-media interaction problems. Similarly, boundary elements would warrant *Generic Boundary Element Processors* for structural and non-structural applications (e.g., BES*, BEF* and BET*). The rationale for such subdivisions is the difference in the high-level command language and data structures natural for each domain (e.g., EF* Processors would probably have `FORM VISCOSITY` and `FORM CONVECTION` commands, respectively; and while BES* Processors would have the same commands as ES* Processors, they would probably output full, non-symmetric assembled matrices rather than a sequence of symmetric element matrices).

---

* The Generic Element Processor design removes the burden of *how* element arrays are to be computed from the algorithm developer, who is primarily interested in *what* should be formed. This (message-passing) feature and the shell/kernel (inheritance) aspect of the design have much in common with so-called object-oriented programming languages, which are intended to simplify and extend the life-cycle of software development efforts [17].

PLANS

CSM Testbed



Having gained some experience with the Generic Element Processor, we believe that it suggests an overall strategy for long-term development of the Testbed. Specifically, we believe that *three* classes of generic Processors should be available: Element, Constitutive and Matrix. This triad can be viewed as a higher-level extension of the truly generic Testbed architecture (NICE) — but with more leverage towards CSM-specific applications. For it is one of the attributes of CSM (say, versus CFD) that it can be neatly partitioned into these three areas. Element developers, matrix algebraists (e.g., solution algorithm developers) and material scientists (e.g., composite mechanicians) can, for the most part, work independently of one another. Yet, all three types of researchers can voluntarily work together — on the same problem — if and when they're ready, via the integration provided by the underlying architecture.

(Imagine a numerical analyst testing a new eigensolution algorithm on a 10×10 manually entered test matrix; a material scientist simultaneously checking out a new high-temperature visco-plastic constitutive model for metal-matrix composites, at a single spatial integration test point; and an element developer independently deriving a new family of hierarchically refinable shell elements validating his/her elements via the MacNeal/Harder test cases — plus others mentioned herein. Next imagine these three researchers stepping aside and letting an engineering analyst apply this triad of combined new capabilities to a complex model of the NASP airframe — without software modifications. This is the envisioned role of the Testbed.)

We are presently working on a plan for the Generic *Matrix* Processor, which would allow an arbitrary number of alternate, or complementary, matrix utilities (and data structures) to be implemented within the Testbed. Importantly, the generic Processor "shell" for these developer-supplied utilities (i.e., "kernels") would feature a common, but extendible, data-structure driven matrix algebraic language, which would enable the user to solve equations via a variety of techniques. This should greatly facilitate the current CSM research in parallel processing — *within* the Testbed.

REFERENCES

[1] Felippa, C.A. and Stanley, G.M., "NICE: A Utility Architecture for Computational Mechanics", in proceedings of *Finite Element Methods for Nonlinear Problems*, eds. Bergan, Bathe and Wunderlich, Europe-US Symposium, Trondheim, Norway, 1985.

[2] Whetstone, W.D., *SPAR Structural Analysis System Reference Manual*, Vol. 1; NASA CR 158970-1, Dec. 1978.

[3] Felippa, C.A., "The Computational Structural Mechanics Testbed Architecture: Volume IV – The Global-Database Manager GAL-DBM", NASA Contractor Report 178387, 1987.

[4] Felippa, C.A., "The Computational Structural Mechanics Testbed Architecture: Volume I – The Language; Volume II – The Directives; Volume III – The Interface", NASA Contractor Report 178384–6, 1987.

[5] Belytchko, T. and Hsieh, B.J., "Non-Linear Transient Finite Element Analysis with Convected Coordinates", *IJNME*, **7**, 255–271, 1973.

[6] Wempner, G., "Finite Elements, Finite Rotations and Small Strains of Flexible Shells", *International Journal for Solids and Structures*, **5**, 117–153, 1969.

[7] Rankin, C.C., "Consistent Linearization of the Element-Independent Corotational Formulation for the Structural Analysis of General Shells", NASA Contractor Report 278428, 1988.

[8] Park, K.C. and Stanley, G.M., "A Curved C^0 Shell Element Based on Assumed Natural-Coordinate Strains", *Journal of Applied Mechanics*, **53**, 278–290, 1986.

[9] Stanley, G.M., "Continuum-Based Resultant Shell Elements", in *Finite Element Methods for Plate and Shell Structures, Vol. 1: Formulations and Algorithms*, eds. T.J.R. Hughes and E. Hinton, pp. 1–45, 1986.

[10] Kang, D.S. and Pian, T.H.H., "A Versatile and Low Order Hybrid Stress Element for General Shell Geometry", *AIAA*, **87-0840**, 633–641, 1987.

[11] Almroth, B.O., Brogan, F.A. and Stanley, G.M., "Structural Analysis of General Shells, Vol. II: User instructions for the STAGS(C-1) computer code", Lockheed Report LMSC-D33873, 1979.

[12] Crisfield, M.A., "A Fast Incremental/Iterative Solution Procedure that Handles Snap-Through", *Computers and Structures*, **13**, 55–62, 1983.

[13] MacNeal, R.H. and Harder, R.L., "A Proposed Standard Set of Problems to Test Finite Element Accuracy", *Journal of Finite Elements in Analysis and Design*, **1**, 3–20, 1985.

[14] Thurston, G.A., Brogan, F.A. and Stehlin, P., "Postbuckling Analysis Using a General Purpose Code", *AIAA Journal*, **24/6**, 1013–1020, 1986.

[15] Starnes, J.H. Jr., Knight, N.F. Jr. and Rouse, M., "Postbuckling Behavior of Selected Flat Stiffened Graphite-Epoxy Panels Loaded in Compression", *AIAA Journal*, **23/8**, 1236–1246, 1985.

[16] Ortiz, M. and Nour-Omid, B., "Unconditionally Stable Concurrent Procedures for Transient Finite Element Analysis", *Computer Methods in Applied Mechanics and Engineering*, **58**, 151–174, 1986.

[17] Cox, B.J., *Object Oriented Programming – An Evolutionary Approach*, Addison-Wesley, 1986.